Pascal 3.2
Workstation System Manual

Vol 2: Programming and Configuration Topics

HP 9000 Series 200/300 Computers

HP Part Number 98615-90023

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

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.

May 1987...Edition 1

September 1987...Updated to include caution against changing HFS default parameters with revision 3.2.

May 1988...Update to include 3.21 revision information, as well as notes about the prefix command's definition with flexible HFS-formatted disc. Also includes the September 1987 update page.

March 1989...Edition 2. This includes the 3.22 revision information.

May 1990...Edition 3. This edition includes additions and changes for the Pascal 3.23 release. The additions to this manual provide information on HP SCSI and Parallel Interfaces.

Table of Contents

Chapter 10: Overview of Workstation Software Features
Introduction 10-1
Chapter Contents 10-1
The Big Picture 10-2
The Series 200/300 Implementation of Pascal 10-4
ANSI/ISO Pascal 10-4
UCSD Pascal Features 10-7
HP Pascal Features 10-8
Series 200/300 Workstation Pascal Compiler Options 10-9
HP Systems Programming Extensions 10-10
HP Series 200/300 Software Libraries 10-11
Library Modules 10-12
User-Designed Modules 10-13

Chapter 11: Data Structures
Data Types 11-1
Scalar Types 11-1
HP Pascal Features 11-2
Packing Variables 11-4
Determining the Size of Variables and Types 11-5
Absolute Addressing of Variables 11-6
Setting a Variable's Absolute Address 11-6
Determining a Variable's Absolute Address 11-6
Conformant Arrays 11-7

Chapter 12: Program Flow
Introduction 12-1
Standard Branching 12-1
CASE/OF 12-1
Procedures and Functions 12-2
Relaxed Typechecking of VAR Parameters 12-2
Procedure Variables and the Standard Procedure CALL 12-4

Chapter 13: Numeric Computation
Introduction 13-1
Numeric Data Types 13-1
Evaluating Scalar Expressions 13-6
The Hierarchy 13-6
Operators 13-8
Numerical Functions 13-12
Dealing with Angles and Such 13-13
Range Limits 13-17
Rounding 13-18
Logarithms and Powers 13-20

Calendar Functions 13-22
The Julian Day 13-22
Number Base Conversion 13-25
Random Numbers 13-28
Workstation Support of Pseudo-Random Numbers 13-28
Using the Pseudo-Random Number Generator 13-29

Chapter 14: String Manipulation
Introduction 14-1
Special Cases of String Assignment 14-1
Declaring String Variables 14-3
String Length 14-3
String Storage in Memory 14-3
String Arrays 14-3
Evaluating Expressions Containing Strings 14-4
Evaluation Hierarchy 14-4
String Concatenation 14-4
Relational Operations 14-5
String Functions 14-6
Substrings 14-6
Current Length of a String 14-7
Maximum Length of a String 14-7
Substring Position 14-7
String-to-Numeric Conversions 14-10
Character-to-Numeric Conversions 14-11
Numeric-to-String Conversions 14-12
Numeric-to-Character Conversions 14-13
String Repeat 14-13
Trimming a String 14-14
Combining Strings 14-14
Reducing Strings 14-15
User-Defined String Functions 14-16
Case Conversion 14-16
String Reverse 14-17
Search-and-Replace Operations 14-18
Sections of Strings 14-19

Chapter 15: Programming With Files
Introduction 15-1
Overview of Files 15-1
Primary versus Secondary Storage 15-1
What Is a File? 15-2
Mass Storage Organization (Non-Hierarchical Directories) 15-2
Classifications of Files 15-4
Item-Oriented Files 15-5
Creating and Writing to an Item-Oriented File 15-5
Reading Sequentially From a File 15-7
Detecting the End of the File 15-7
Line-Oriented (Text) Files 15-9
Creating a File 15-9

Writing to a File 15-11
Reading a Text File with the Editor 15-12
Reading a Text File with a Program 15-12
Detecting the End of the File 15-14
Detecting the End of a Line 15-14
Other Types of Text Files 15-15
More Details on Programming With Files 15-20
Creating New Files 15-20
File Position 15-22
The Buffer Variable 15-22
File States 15-22
Pascal Primitive File Operations 15-23
Restrictions on APPEND 15-25
Disposing of Files 15-25
Opening Existing Files 15-25
Sequential File Operations 15-26
Direct Access (Random Access) Files 15-28
Text Files INPUT and OUTPUT 15-30
Representations of a Text File 15-31
Formatted Input and Output 15-33
Reading a STRING or PAC from a Text File 15-34
RESET, REWRITE, OPEN, and APPEND 15-34
SRM Concurrent File Access 15-37
SRM Access Rights 15-39
HFS Permissions 15-40
Debugging Programs Which Use Files 15-40
How Magnetic Discs Work 15-41
No Room on Volume 15-42

Chapter 16: Dynamic Variables and Heap Management
Stack/Heap Architecture 16-1
Dynamic Variables and Pointers 16-1
Heap Management 16-2
MARK and RELEASE 16-2
DISPOSE 16-3
Mixing DISPOSE and RELEASE 16-4

Chapter 17: Error Trapping and Simulation
Introduction 17-1
Error Trapping and Simulation 17-1
The IORESULT Function 17-2
$IOCHECK$ and IORESULT 17-3
Extended Error Information 17-5
Determining a File's Existence 17-7
Error Simulation 17-9

Chapter 18: Special Configurations
Introduction 18-1
Chapter Organization 18-1
The Booting Process 18-4

The Boot ROM 18-4
The Pascal System Discs 18-5
The System Boot File (SYSTEM_P) 18-5
The Initialization Library (INITLIB) 18-5
The Command Interpreter (STARTUP) 18-6
The Auto-Configuration Program (TABLE) 18-7
The AUTOSTART and AUTOKEYS Stream Files 18-7
Libraries 18-7
The Auto-Configuration Process 18-8
The Unit Table 18-8
How Unit Numbers Are Assigned 18-9
Unblocked Devices 18-10
Blocked Devices 18-10
Choosing the System Volume 18-13
Failure of the TABLE Program 18-13
Example Special Configurations 18-14
Hard Disc Partitioning 18-14
Multiple On-Line Systems 18-15
Adding Interfaces and Peripherals 18-16
Changing the System Printer 18-20
Using Bubble and EPROM Cards 18-21
Using Alternate DAMs 18-22
Modifying the Configuration 18-27
Coalescing Hard Disc Volumes (LIF Only) 18-27
Copying System Files and Changing Their Names 18-35
AUTOSTART and AUTOKEYS Stream Files 18-38
Adding Modules to INITLIB 18-39
Modifying the TABLE Program 18-51
Commentary on the CTABLE Program 18-52
Modifying Module OPTIONS 18-53
About Module CTR 18-64
About Module BRSTUFF 18-65
About Module SCANSTUFF 18-65
About Module SCSIscanstuff 18-65
Discussion of the Main Body of CTABLE 18-65
Editing CTABLE 18-70
Compiling and Running CTABLE 18-71
Verifying the New Configuration 18-71
Making the New Configuration Permanent 18-72
Setting Up an SRM System 18-74
Example SRM Configuration 18-74
Prerequisites 18-75
Overview of SRM Installation 18-75
Installing the SRM Driver Modules 18-76
Re-Configuring with TABLE 18-76
Creating the Required Directories and Files 18-77
Copying the System Files to SRM 18-80
Adding Modules to INITLIB 18-82
Replacing INITLIB 18-83
Multi-Disc SRM 18-85

Chapter 19: Non-Disc Mass Storage

Introduction 19-1
Summary of Configuration Modifications 19-1
Mass Storage Comparison 19-2
Using Bubble Cards 19-3
Power Constraints 19-3
Bubble Card Configuration 19-3
INITLIB Driver Modules 19-5
CTABLE Modifications 19-8
Compiling CTABLE 19-9
Linking CTABLE 19-9
Bubble Cards in the File System 19-10
Initialization 19-11
Interrupts and Overlapped I/O 19-11
Using EPROM Memory 19-12
Overview 19-12
Configuration Changes Required 19-13
INITLIB Driver Modules 19-13
Programmer Card Installation 19-14
EPROM Card Installation 19-16
The Programming Utility 19-19
Transferring Volumes to EPROM 19-20
Transferring Files to EPROM 19-21
The EPROM Transfer Utility 19-23
Loading the EPROMS Module 19-30
CTABLE Modifications 19-32
EPROM Cards in the File System 19-34
Using Cartridge Tapes 19-35
Tape Drives Supported 19-35
Tape Access Methods 19-35

Chapter 20: Backup Utilities

Introduction 20-1
The Backup Utility 20-1
The Tape Backup Utility 20-1
Using the Backup Utility 20-2
Using the Tape Backup Utility 20-9
Using the File System for Direct Tape Access 20-13

Chapter 21: HFS Setup and Utilities

Setting Up An HFS System 21-1
General Procedure 21-1
The MKHFS Utility 21-2
Using MKHFS 21-2
OSINSTALL Utility 21-5
HFS Booting Overview 21-5
Using OSINSTALL 21-6
Check 21-6
Install 21-8
Order 21-8

Remove 21-9
Zero 21-9
The HFSCK Utility 21-10
Why Should I Need To Run This Utility? 21-10
Invoking the HFSCK Utility 21-11
HFSCK Confirmation Requests 21-12

Chapter 22: Porting to Series 300
Introduction 22-1
Who Needs this Information? 22-1
Methods of Porting 22-1
Chapter Organization 22-1
Description of Series 300 Enhancements 22-2
Areas of Change 22-2
Areas that Did Not Change 22-2
Displays 22-3
Processor Boards 22-4
Battery-Backed Real-Time Clock 22-5
Built-In Interfaces 22-5
ID PROM 22-7
Just Loading and Running Programs 22-8
Should Problems Arise 22-8
Using a Configuration Program 22-9
Example of Serial Interface Configuration 22-9
Using Compatibility Hardware 22-10
Hardware Description 22-11
Steps in Using this Card Set 22-12
Modifying the Source Program 22-13
Programs Compiled on Pascal 2.1 (or Earlier Versions) 22-13
HP 98203 Specific Key Codes 22-14
Linked-In, Incompatible Modules 22-14
Use of Low-Level Procedures 22-14
Full Utilization of Series 300 Hardware Features 22-14

Appendix A: Error Messages
Unreported Errors A-2
Boot-Time Errors A-2
Run-Time Errors A-3
I/O System Errors A-4
I/O Library Errors A-6
Graphics Errors A-7
Loader/SEGMENTER Errors A-8
SEGMENTER Errors A-8
Loader Boot-Time Errors A-8
Pascal Compiler Errors A-9
Compiler Options A-11
Implementation Restrictions A-11
Non-ISO Language Features A-12
Assembler Errors A-13
Debugger Error Messages/Conditions A-14
VMELIBRARY Errors A-16

Appendix B: Technical Reference

System History B-1
Pascal 1.0 ..... B-1
Pascal 2.0 and 2.1 ..... B-2
Pascal 3.0 ..... B-5
Pascal 3.01 ..... B-9
Pascal 3.1 ..... B-10
Pascal 3.12 ..... B-14
Pascal 3.2 ..... B-14
Pascal 3.21 ..... B-16
Pascal 3.22 ..... B-17
Pascal 3.23 ..... B-18
File Interchange Between Pascal and BASIC B-19
Module Names Used by the Operating System B-20
Physical Memory Map B-21
16 Megabyte Address Range B-21
The Overall ROM Memory Map B-22
Memory Mapped I/O B-22
External I/O B-22
Internal I/O B-23
The Software Memory Map B-24

Appendix C: Character Sets

U.S. ASCII Character Set C-2
U.S. ASCII Character Set C-3
U.S./European Display Characters C-4
U.S./European Display Characters C-5
U.S./European Display Characters C-6
U.S./European Display Characters C-7
U.S./European Display Characters C-8
U.S./European Display Characters C-9
Katakana Display Characters C-10
Katakana Display Characters C-11
Monochrome Highlight Characters C-12
Color Highlight Characters C-13

Appendix D: Command Summaries

Main Command Level Summary D-1
Editor Command Summary D-2
Text Modifying Commands D-2
Text Formatting Commands D-2
Miscellaneous Commands D-2
Cursor Keys D-3
Cursor Positioning Commands D-3
Filer Command Summary D-4
Volume Related Commands D-4
Exit Commands D-4
File Related Commands D-5
Workfile Related Commands D-5
Librarian Command Summary D-6

General Commands D-6
Copy Mode Commands D-6
Edit Mode Commands D-6
Link Mode Commands D-7
Unassemble Commands D-7
Debugger Command Summary D-8

Appendix E: Glossary E-1

Overview of Workstation Software Features 10

Introduction

This chapter briefly lists the features of the Pascal language implemented on the Series 200/300 Workstation System. It also briefly describes the Procedure Library supplied with the system.

Chapter Contents

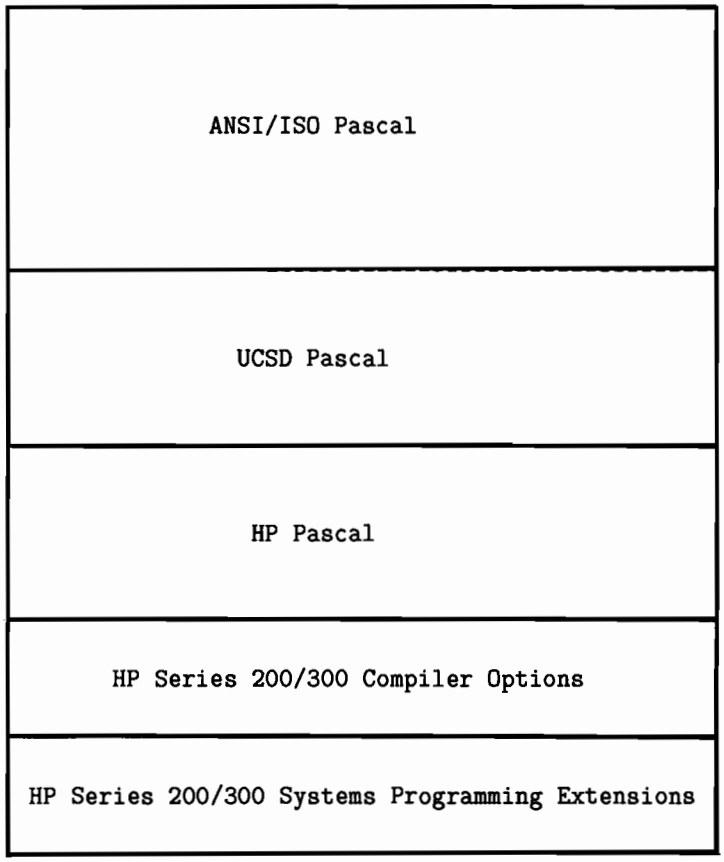
- Constituents of the HP Series 200/300 Pascal language implementation
- ANSI/ISO Pascal features
- UCSD Pascal[1] extensions
- HP Pascal extensions
- HP Series 200/300 Pascal Compiler options
- HP Series 200/300 Systems Programming extensions
- Overview of HP Series 200/300 Workstation software libraries

[1] UCSD Pascal is a trademark of the Regents of the University of California.

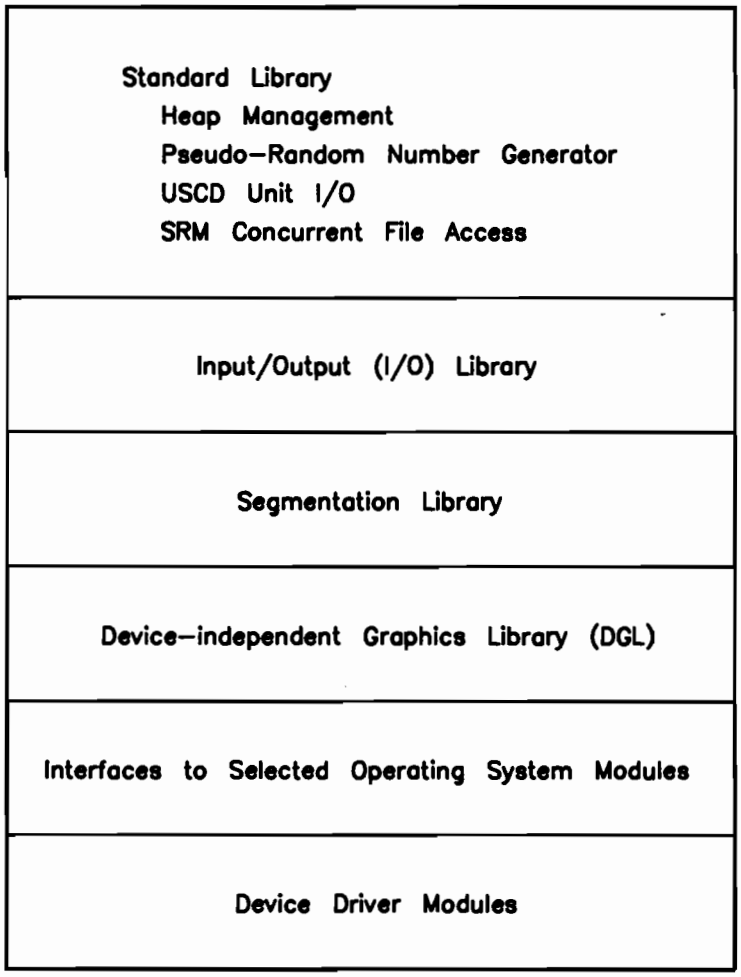
The Big Picture

The software supplied with the HP Series 200/300 Workstation Pascal system can be divided into several components, as shown in the following diagrams:

Components of the Series 200/300 Implementation of Pascal



The HP Series 200/300 Software Libraries

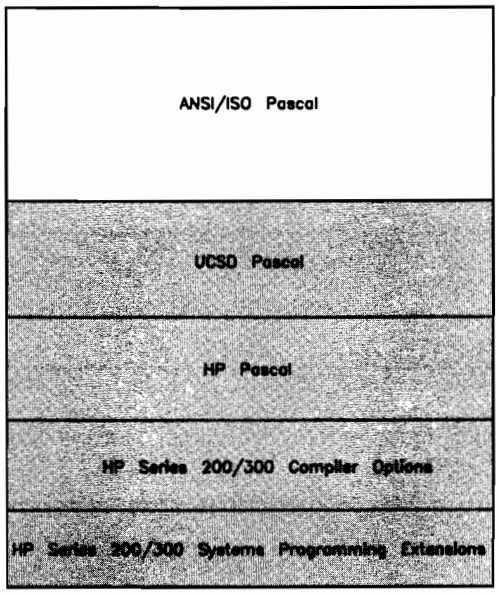


Subsequent sections of this chapter further describe each part of the drawings.

The Series 200/300 Implementation of Pascal

The HP Series 200/300 Workstation implementation of the Pascal language contains a full complement of features. This section describes the components of this implementation of the Pascal language.

ANSI/ISO Pascal



The term "ANSI/ISO" is an abbreviation of for two different Pascal standards. The "ANSI" portion stands for the Pascal standard adopted jointly by ANSI (the American National Standards Institute) and IEEE (the Institute of Electronics and Electrical Engineers). The "ISO" portion stands for the Pascal "Level 1" standard adopted by ISO (the International Standards Organization).

The HP Series 200/300 Workstation Pascal implementation contains **all** of the features of both the ANSI/IEEE and the ISO Pascal standards. Programming in ANSI/ISO Pascal is described in the *Programming and Problem Solving with Pascal* textbook supplied with the Workstation Pascal system.

Here is a list of the keywords in ANSI/ISO Pascal, which are all supported in this implementation.

Declarative Statements

```
program
const
label
type
var
procedure
function
```

Program Parameters

```
input
output
```

Data Types

```
array
boolean
char
file
integer
packed
real
record
set
text
with
```

Program-Flow Control

```
begin...end
case...of
if...then...else
goto
for...to
   ...downto
repeat...until
while....do
```

Standard Procedures

```
get
new
pack
page
put
read
readln
reset
rewrite
unpack
write
writeln
```

Pre-defined Constants

```
false
true
nil
maxint
```

Numeric Functions

```
abs
arctan
cos
exp
ln
odd
round
sin
sqr
sqrt
trunc
```

File Functions

```
eof
eoln
```

Ordinal Functions

```
chr
ord
pred
succ
```

Assignment Operator

```
:=
```

Arithmetic Operators

```
+
-
*
/
div
mod
```

Comparison Operators

```
<
<=
=
>=
>
<>
```

Logical Operators

```
and
not
or
```

Set Operators

```
+
-
*
in
```

Extending "Standard" Pascal's Capabilities

Pascal is a general-purpose programming language. It was originally designed as a language to teach structured programming, and it has since gained widespread use due to this orientation. However, as with all languages, the Pascal programming language cannot satisfy *every* programmer's needs; in such cases, the feature set can be "extended" to fit certain applications.

There are two general ways to extend the capabilities of a programming language:

- Add extensions to the language itself.
- Write "library" routines that can be called from the language.

The Pascal Workstation designers have used *both* methods to add capabilities to this system. Language extensions are described in the following sections, followed by libraries in later sections.

Language Extensions

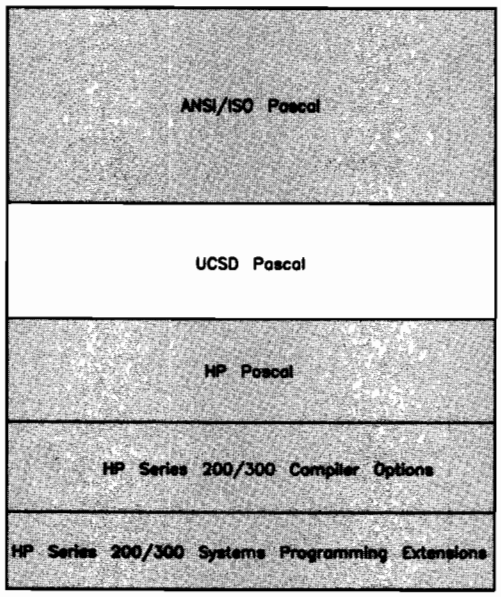
Adding extensions to a language requires that the designers add to the list of "keywords" that the Compiler will recognize. This Pascal implementation contains four general categories of extensions:

- UCSD Pascal[1] extensions
- HP Pascal extensions
- Series 200/300 Workstation Pascal Compiler options
- HP Series 200/300 Systems Programming extensions

Each category is further described in subsequent sections.

[1] UCSD Pascal is a trademark of the Regents of the University of California.

UCSD Pascal Features



UCSD Pascal adds many useful features to the "standard" Pascal language. The UCSD Pascal features which this Pascal implementation supports are fully described in "Supported Features of UCSD Pascal" in the "Workstation Implementation" appendix of the *HP Pascal Language Reference*. Here is a brief summary, showing the various levels of support for UCSD features.

Fully Supported

`blockread`
`blockwrite`
`close`
Compiler options
`external`
Files
`fillchar`
Heap Management
`moveleft`
`moveright`
Reals
`scan`
`set`
`sizeof`
Special Program Heading
Standard Units
Strings
`unitbusy`
`unitclear`
`unitread`
`unitwrite`
Untyped Files

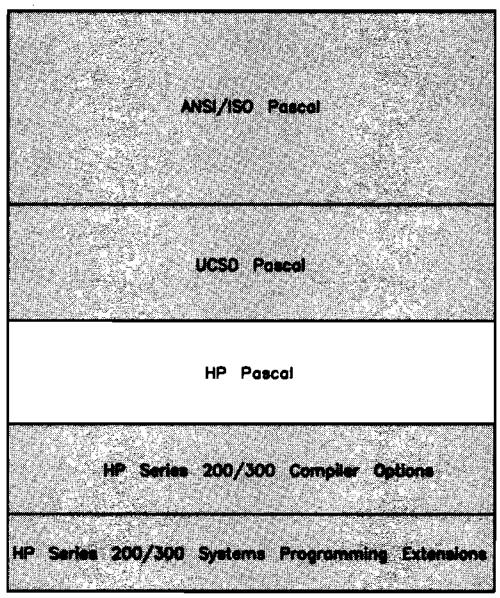
Slight Differences

`CASE`
Comments
Compilation Units
`exit`
`gotoxy`
`halt`
16-bit Integers
`interactive`
`ioresult`
`memavail`
`seek`
`time`
Type Checking
`unit`

Unsupported Features

`log`
Long integers
Multi-word comparisons
`pwroften`

HP Pascal Features

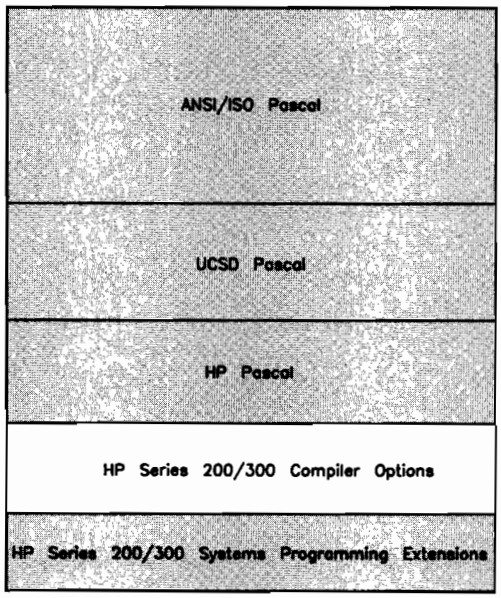


HP Pascal includes many of the UCSD extensions to ANSI/ISO Pascal, plus some of its own. The HP Pascal extensions to ANSI/ISO Pascal are briefly summarized in this section, and more fully described at the **beginning** of the *HP Pascal Language Reference* manual. Complete, detailed descriptions of individual procedures, reserved words, etc., are provided in the **body** of the same reference. Examples of using many of these HP Pascal extensions are provided in subsequent chapters of this manual. Here is a brief summary of the areas in which HP Pascal has extensions:

HP Pascal Features

Constant Expressions
Early Program Termination
Extended Variable Assignment Compatibility
Full File-I/O Feature Set
Compiler Options
Functions May Return Any Structured Type
Heap Management Capabilities
Identifiers May Contain "_" Character
Intermixing of Declaration Parts of Programs
Longreal Data Type
`minint` Pre-defined Constant
Modules
Numeric-to-String Conversions
`OTHERWISE` in `CASE` Statement
Record List in `WITH` May Include Function Calls
Record Variants May Be Subranges
String Literals May Contain Control Characters
String Data Type
Structured Constants
Conformant Arrays
Floating point Math Optimizations

Series 200/300 Workstation Pascal Compiler Options



Some Compiler options affect the way that the Compiler emits object code, while others allow the use of UCSD and HP Series 200/300 Systems Programming extensions. For a description of each option, refer to the "Series 200/300 Compiler Options" section of the "Workstation Implementation" appendix of the *HP Pascal Language Reference* manual.

Code-Generation Control

```
callabs
code
code_offsets
debug
float_hdw
heap_dispose
if
iocheck
ovflcheck
partial_eval
range
stackcheck
```

Message Control

```
ansi
copyright
warn
```

Compiler Listing Control

```
linenum
lines
list
page
pagewidth
tables
```

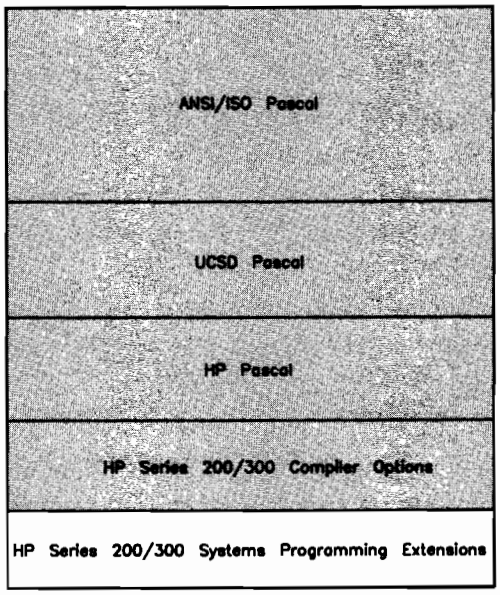
Use of External Files

```
def
include
ref
search
search_size
```

Language Feature Control

```
alias
allow_packed
save_const
switch_strpos
sysprog
ucsd
```

HP Systems Programming Extensions



The HP Series 200/300 Systems Programming extensions are briefly described in the following list. Programming examples of most features are given in subsequent chapters of this manual. Complete descriptions of all features are provided in the "Systems Programming Extensions" section in the "Workstation Implementation" appendix of the *HP Pascal Language Reference* manual.

Error Trapping and Simulation

```
escape
escapecode
ioresult
try/recover
```

Absolute Address of Variables

```
addr
var syntax
```

Size of Variables and Types

```
sizeof
```

Relaxed Type-Checking

```
anyptr
anyvar
```

Special Procedure Calls

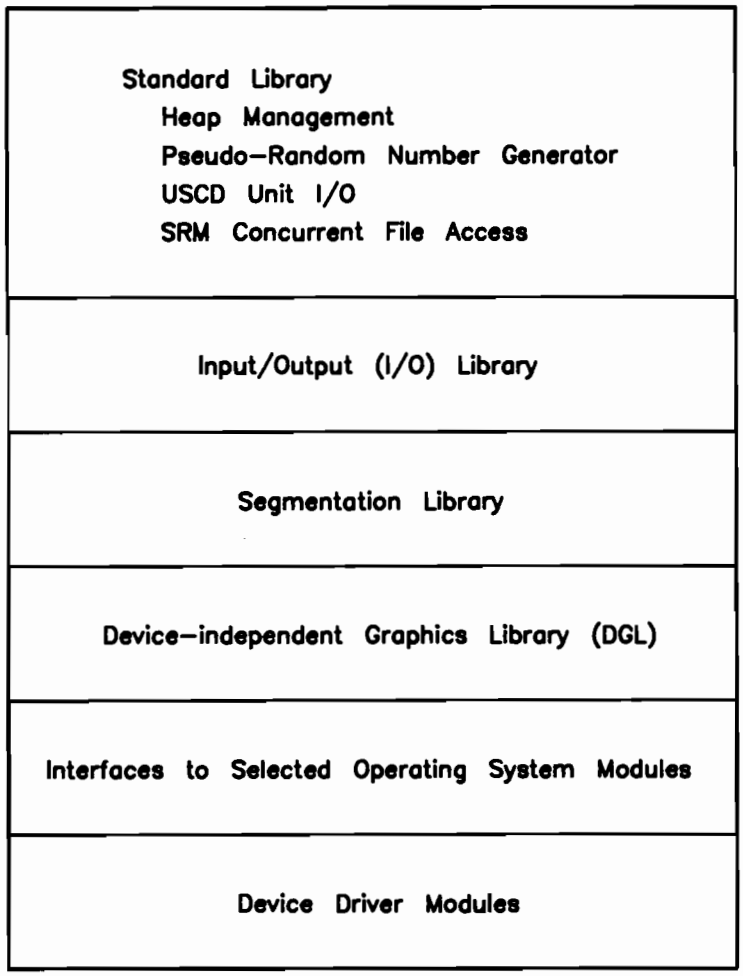
```
call
```
Variables of type `procedure`

With the power of these System Programming features, however, comes the restriction that programs that use them will probably be dependent upon the Workstation Operating System and possibly the hardware on which the programs are executed (which may include its specific configuration).

A Final Word Concerning Language Extensions

Although these extensions provide many additional capabilities to the Pascal language, they do not provide a full set of tools for accessing Workstation computer capabilities. That tool set is provided by software libraries.

HP Series 200/300 Software Libraries



The second way to "extend" a language's capabilities is to place commonly used procedures, functions, data types, and so forth into "libraries" which are accessible to all programmers on the system. In this system, these libraries consist of object-code "modules" produced by the Compiler, Assembler, or Librarian. Each module is an *independent* program fragment that contains data and/or procedures which are usable by other programs (and other modules). The general topic of modules is discussed in the Compiler, Assembler, and Librarian chapters of this manual.

Library Modules

The following list of libraries is organized according to the file in which they are shipped with the system. A list of the discs and files upon each is provided in the *Pascal User's Guide*; you may also want to generate your own list by using the Filer's List (or Extended_list) command.

The LIBRARY file provides the following four modules:

- The HPM (Heap Management) module provides the **new** and **dispose** procedures that can be used to allocate and reclaim memory used by dynamic variables. See the "Dynamic Variables and Heap Management" chapter for examples.
- The RND (Random Numbers) provides the **random** procedure and the **rand** function that are used for generating pseudo-random sequences. See the "Numeric Computation" chapter for examples.
- The UIO (UCSD Unit I/O) module provides the **blockread**, **blockwrite**, **unitbusy**, **unitclear**, **unitread**, **unitwait**, **unitwrite** procedures that are used for "low-level" input and output (I/O) operations with mass storage "blocks". See "Supported Features of UCSD Pascal" in the "Workstation Implementation" appendix of the *HP Pascal Language Reference*.
- The LOCK module provides features that support concurrent file access on the Shared Resource Manager (SRM) "file server" system. The **lock** function and the **unlock** and **waitforlock** procedures are used to lock and unlock shared files. See the "Programming with Files" chapter for examples.

The IO (Input/Output) file provides several modules which provide constants, types, variables, procedures, and functions used for communicating with non-filesystem devices through HP Series 200/300 interfaces. These procedures are described in several chapters of the *Pascal Procedure Library* manual. A functionally grouped list of all procedure and functions is also provided at the beginning of the "Procedure Reference" section of that manual.

The "DGL" (Device-independent Graphics Library) files include GRAPHICS, FGRAPHICS, and FGRAPH20. These files contain the modules that provide procedures and functions for drawing and labeling graphics images on both raster and physical-pen plotting devices. They also contain procedures and functions for graphics input devices, such as graphics tablet, mouse, knob, or TouchScreen. See the *Pascal Graphics Techniques* manual for examples.

The SEGMENTER file provides procedures and functions for executing small segments of larger programs, in order to decrease memory requirements. These procedures are described in the "Segmentation" chapter of the *Pascal Procedure Library* manual.

The INTERFACE file provides an interface to selected Operating System modules. Here are the modules which are documented in the manuals:

- IODECLARATIONS and various other modules provide many useful data structures that are used by various parts of the system and by many procedure libraries. See the "Introduction to I/O" chapter of the *Pascal Procedure Library* manual for details on the IODECLARATIONS module.

- The SYSDEVS (System Devices) module provides data structures, procedures and functions for using the built-in displays, keyboards, and timers of Series 200/300 machines. These procedures are described in the "System Devices" chapter of the *Pascal Procedure Library* manual.

Note

Of these Operating System modules with interfaces in the INTERFACE module, *only* the use of the IODECLARATIONS and SYSDEVS modules are documented in the *Pascal Procedure Library* manual.

The VMELIBRARY file provides data transfer capabilities between the Series 300 and the HP98646A VMEbus Interface. This will allow you to add a larger variety of peripherals to your system. For more information, see the VME chapter of the *Pascal 3.2 Procedure Library.*

The SYSBOOT file contains a programmable, callable function that will cause a system to boot or reboot. For more information see the *Pascal 3.2 Procedure Library.*

The SCSILIB file provides programmatic access to a SCSI bus attached to the Series 300 SPU via the HP 98658A or HP 98265A SCSI interface. For more information see the chapter "SCSI Programmer's Interface" in the *Pascal 3.2 Procedure Library.*

Other files on the ACCESS: disc (CONFIG: or LIB: disc if the workstation was purchased on single-sided media) provide device driver modules which contain code that the system uses to communicate with with interfaces and devices. For instance, the GPIO file (which contains a module of the same name) provides driver routines that are used to communicate through an HP 98622 General-Purpose Input/Output (GPIO) interface. Note that these device driver files contain no "export text" that describes the procedures, etc. in the modules, because the drivers don't need it.

The most commonly used modules are automatically loaded into your system during the booting process, because they are in the INITLIB file on the BOOT: disc (or BOOT2: disc). For instance, the CS80 module provides routines which communicate with CS/80 and SS/80 type disc drives. The "Special Configurations" chapter of this manual describes the booting process and all driver modules shipped with the system.

The driver that provides access to Hierarchical File System (HFS) discs, which are Series 300 HP-UX compatible, is found on the HFS: disc (HFS1: disc if the system came on single-sided media). Access to HFS discs is most convenient when this driver is installed in INITLIB.

You may have had to install other driver modules yourself while configuring your system. The description of this process is in the "Adding Peripherals" section of the *Pascal User's Guide.*

User-Designed Modules

One of the most powerful capabilities of this system is that you can design your own specialized libraries using the HP Pascal **module** construct. That subject is discussed in the "Compiler," "Assembler," and "Librarian" chapters of this manual, as well as in the "Overview" chapter of the *Pascal Procedure Library* manual.

Data Structures 11

One of the most powerful features in Pascal is the ability to create *data structures.* A data structure is an arrangement of types of data in such a way that it most accurately represents the model you are trying to represent.

Data Types

Hewlett-Packard Workstation Pascal supports all standard Pascal data types. This section briefly summarizes these types and how they are used. Extensions to standard Pascal provided by Workstation Pascal will be noted in the text.

Scalar Types

The word "scalar" in the phrase "scalar types" means *single-valued;* that is, variables of these types each contain only one piece of data. This is opposed to the concept of "structured" types, a kind of do-it-yourself data type. Structured types are covered later in the chapter.

Standard Data Types

The simplest data structures are those simple, standard types provided by the Pascal language. On the Series 200 and 300 computers, these are:

`integer`	A 32-bit signed integer number.
`real`	A 64-bit signed floating-point number.
`char`	An 8-bit ASCII character.
`boolean`	True or false values.

Note that the types above are implementation-dependent in the areas of number of bits, format of bits, etc.

HP Pascal Features

These are features unique to HP Pascal and may not port to other implementations of Pascal.

Array Constants

To make a constant which is of an array type, you specify the type, a "[", the values, separated by commas, and a "]". The base type must be declared before the constant declaration; e.g., in the sample code below, you must declare a type **MonthsType** before you can declare the 12-element constant array **DaysPerMonth**. For example:

```
type
  MonthsType=        array [1..12] of integer;
const
  DaysPerMonth=      MonthsType[31,28,31,30,31,30,31,31,30,31,30,31];

type
  LineType=          array [1..4] of real;
  MatrixType=        array [1..4] of LineType;
const
  Identity=          MatrixType[LineType[1.0, 0.0, 0.0, 0.0],
                               [LineType[0.0, 1.0, 0.0, 0.0],
                               [LineType[0.0, 0.0, 1.0, 0.0],
                               [LineType[0.0, 0.0, 0.0, 1.0]];
```

Note

When making a structured constant of a multidimensional array, it must be declared one dimension at a time; e.g., a vector of vectors, rather than a 2D array.

When declaring an array constant and there are several identical values in consecutive places, you can declare them something like this:

```
type
  VectorType=        array [1..100] of integer;
const
  Vector=            VectorType[1,2,3,6 of 0,7,90 of 0];
```

This results in an array, none of whose elements' values can be changed, in which there are these values:

1, 2, 3, six zeroes, 7, and ninety more zeroes.

Record Constants

When declaring a constant of some record type, you must specify the field name before the corresponding value. For example:

```
type
  DateType=           record
                        Month:     1..12;
                        Day:       1..31;
                        Year:      integer;
                      end;
  MaritalStatusType= (Single, Married, Separated,Divorced, Widowed);
  IntervieweeType=    record
                        Name:              string[30];
                        BirthDate:         DateType;
                        MaritalStatus:     MaritalStatusType;
                        NumberOfChildren: 1..20;
                        Education:         set of (HighSchool, BA, BS, MA,
                                                   MS, PhD, DD);
                      end;
const
  ThisPerson=         IntervieweeType[Name:             'John Q. Public',
                                      BirthDate:        DateType[Month: 10,
                                                                 Day:   20,
                                                                 Year:  1955],
                                      MaritalStatus:    Married,
                                      NumberOfChildren: 1,
                                      Education:        [HighSchool, BS]];
```

Set Constants

Set constants can be made. In a set constant, the elements must be surrounded by brackets, so the compiler knows that it is a set constant. No base type is required in order to declare a set constant:

```
const
  Vowels=             ['a','e','i','o','u'];  {set constant}
```

Packing Variables

Algorithms that depend on specific packed (or unpacked) sizes, addresses, etc. will probably not port, even to other HP Pascal implementations.

In the discussion of arrays, there was some mention of PACs, or packed arrays of characters. Arrays of ⟨*type*⟩ are different than packed arrays of ⟨*type*⟩, no matter what ⟨*type*⟩ is.

Probably the most common reason for packing a data structure is that it sometimes takes less memory to contain the data. However, you may pay for the reduced memory demands in *increased* time required to access the variables. There is a way, though, to get the best of both worlds: the lower memory consumption of packed variables and the high speed of unpacked variables. The **pack** and **unpack** procedures allow you to do this.

Suppose you have an array of packed variables. They are packed to save file space and memory space. You can unpack an element, process it at the higher speed afforded by unpacked variables, and repack it.

One precaution: when packing variables, you may not get exactly what you wanted. The compiler may do some field justification to byte- or nybble boundaries in order to make processing faster.

```
$tables$
   . . .
type
  NotReallyPackedRec=   packed record
                          case integer of
                            1: (RealNumber:  real);
                            2: (SignBit:     boolean;
                                Exponent:    packed array [1..11] of boolean;
                                Mantissa:    packed array [1..52] of boolean;)
                        end;
```

One would think that this is a convenient way to deal with the various subfields in a real number. However, although you specify "**packed**", it is not really packed; it's no more compact than if you hadn't specified **packed**. You can verify this by specifying the compiler option **$tables$**, which causes the following information (among other things) to be printed in the listing:

```
NOTREALLYPACKEDREC type
 record unpacksize=11 align=2
   EXPONENT          field offset=2
   MANTISSA          field offset=4
   REALNUMBER        field offset=0
   SIGNBIT           field offset=0 bitoffset=0
```

The reason that the above example is not any more compact that the unpacked version is that there are some constraints on packing. While an attempt is made by the compiler to use less space, there are also some requirements for efficient access; both packing *and* alignment are considerations:

- Nothing whose size is a long word or more is packed. Thus, integers, pointers, and reals are not packed. Also, if a record contains other records, the internal records are not packed with respect to the record which contains them.
- Everything that is packed must be accessible with one long-word access, and long-word accesses must take place on even-byte boundaries. For example, it is conceivable that a 17-bit field would not be accessible by a single 32-bit access. Thus, this field would not be packed in this way.
- Arrays are packed along 1, 2, 4, 8, or 16-bit boundaries. Thus, an array of 5-bit fields would be packed only to 8-bit boundaries.

In addition to the optional keyword **packed** in front of **array**, **record**, **set**, and **file** type specifiers, there are two routines, **pack** and **unpack**, which convert an **array of** ⟨*type*⟩ to a **packed array of** ⟨*type*⟩ and vice versa. See the *HP Pascal Language Reference* for details on **pack** and **unpack**.

Determining the Size of Variables and Types

Algorithms that depend on specific packed (or unpacked) sizes, addresses, etc. will probably not port, even to other HP Pascal implementations.

The size (in bytes) of a data type or variable can be determined by the Systems Programming function[1] **sizeof**. To use this feature, you must use the **$sysprog$** or the **$ucsd$** Compiler option. Here are examples of usage:

```
$sysprog$
  . . .
VBytes:=sizeof(Variable);
TBytes:=sizeof(TypeName);
```

If the variable or type is a record with variants, optional tagfield constant(s) may follow the variable name parameter. Syntactically, it is similar to a call to the standard Pascal procedure **new**:

```
NBytes:=sizeof(RecVar,TrueField,BlueField);
```

The **sizeof** function cannot be used to determine the size of elements of packed structures, unless the **$allow_packed$** directive is enabled. Using **sizeof** to determine the size of elements of packed structures is not recommended; it may not return the correct value for certain packings.

[1] Although **sizeof** looks like a function, it really is not one; it is actually a form of compile-time constant.

Absolute Addressing of Variables

Systems Programming extensions also provide the capability of programmatically setting and determining the absolute address of variables. This capability requires **$sysprog$**.

Algorithms that depend on specific packed (or unpacked) sizes, addresses, etc. will probably not port, even to other HP Pascal implementations.

Setting a Variable's Absolute Address

A variable may be declared as located at an absolute or symbolically named address:

```
var
  SysFlag[hex('FFFFFED2')]:              char;
  AssemblerSymbol['external_name']:      integer;
```

Each variable named in a declaration may be followed by a bracketed address specifier. An integer constant gives the absolute address of the variable. A quoted string literal gives the name of a load-time symbol which will be taken as the location of the variable; such a symbol must be present in RAM when the program is loaded. These variables are not accessed as globals and do not count against the 32K-byte limit per module or the 64K-byte total limit.

Determining a Variable's Absolute Address

Algorithms that depend on specific packed (or unpacked) sizes, addresses, etc. will probably not port, even to other HP Pascal implementations.

The **addr** function returns the address of a variable in memory as a value of type **anyptr**. This also requires **$sysprog$**.

```
type
  SomeType=           ⟨type_declaration⟩;
var
  Pointer1,Pointer2: anyptr;
  Variable1:         integer;
  Variable2:         SomeType;
      .
      .
      .
  Pointer1:=addr(Variable1);
  Pointer2:=addr(Variable2,Offset);
```

The **addr** function accepts, as an optional second parameter, an integer "offset" expression which will be added to the address; this has the effect of pointing "offset" bytes away from where the variable begins in memory. A positive offset addresses bytes higher in memory, and a negative offset addresses bytes lower in memory.

The **addr** function is primarily used for building or scanning data structures whose shapes are defined at run-time rather than by normal Pascal declarations.

Note

Programs using this feature must be *very carefully debugged.* Careless use of the pointers returned by **addr** can crash your system.

The **addr** function has the same dangers described above for **anyptr**s, in addition to some of its own. Use of the "offset" can produce a pointer to almost anywhere, with concomitant dangers to the integrity of system memory.

Never use **addr** to create non-local pointers to the local variables of a procedure or function. Storage for local variables is recovered when the routine exits, so the value returned by **addr** is ephemeral.

Conformant Arrays

Conformant arrays are arrays in a called routine which automatically conform to the size of the array which was passed to the routine. The conformant array feature allows arrays of various sizes to be passed to a single formal parameter of a routine. It also provides a mechanism for determining at runtime the indices with which the actual parameter was declared.

Conformant arrays are defined within the formal parameter list of a procedure or function. They may be passed by value or by reference.

Conformant arrays may be packed or unpacked. Their organizations, or representations are defined by "schemas." Unpacked schemas may have any number of indices, whereas packed schemas are limited to one index. In a schema with multiple indices, the final array definition may be either packed or unpacked. Conformant arrays may not be packed arrays of characters (PAC) types.

An abbreviated syntax is allowed for specifying multi-dimensional conformant arrays. The schema:

```
array [⟨index type⟩] of
  array [⟨index type⟩] of
                .  .  .
  array [⟨index type⟩] of ⟨type id⟩
```

can be written as:

```
array [⟨index type⟩;
       ⟨index type⟩;
                .  .  .
       ⟨index type⟩] of ⟨type id⟩
```

The bound identifiers (the low bound identifier and the high bound identifier in the index type specification) are used to determine the indices of the actual parameter passed to the formal conformant array. Their values are set when the routine is entered, and they remain constant throughout that activation of the routine.

Bound identifiers are special objects. They are not constants and they are not variables; thus, they cannot be used in **const** or **type** definitions, and may not be assigned to, or used in any other context in which a variable is assigned to (argument to **var** parameter, **for**-loop control variable, etc).

Conformability

An actual array parameter must "conform" to the corresponding formal parameter. That is, an array variable may be passed to a routine with a corresponding formal conformant array parameter if the array variable's type "conforms with" the schema of the formal parameter.

An informal way of describing conformability is to say that the array variable's type conforms with schema if, for each dimension of array type and schema, the index types and component types of array type and schema "match."

For instance, given the following types and conformant array schemas:

Types:

```
type
  Index=      1..20;
  T1=         packed array [1..10] of integer;
  T2=         array [1..5, 1..10] of integer;
  T3=         array [1..50] of integer;
```

Conformant Array Schemas:

Schema 1: `array [lo..hi: Index] of array [smallest..largest: Index] of integer;`

Schema 2: `packed array [little..big: Index] of integer;`

Schema 3: `array [least..greatest: Index] of integer;`

Schema 4: `array [lo..hi: Index; lo2..hi2: Index] of integer;`

The following relationships are true:

- Type `T1` conforms with Schema 2 only.
- Type `T2` conforms with Schemas 1 and 4 only.
- Type `T3` does not conform with any of the schemas (because 50 is greater than the maximum value for type Index).

Equivalence

Two conformant array schemas are "equivalent" if all of the following are true:

- The ordinal type identifier in each corresponding index type specification denotes the same type.
- Either:
  - the type identifier of the two schemas denotes the same type, or
  - the component conformant array schemas of both schemas are equivalent.

Congruency

An actual array parameter of an actual procedure or function parameter must be "congruent" with the corresponding formal parameter. Two conformant array schemas are "congruent" if all of the following are true:

- The two schemas are both packed or unpacked.
- The two schemas are both by-value or by-reference schemas.
- The two schemas are equivalent.

An example of where you would be able to use conformant arrays to your advantage is shown below. Suppose you need a vector (a one-dimensional array) where the first element of the array equals 1, the second element equals 2, etc. With a procedure which uses conformant arrays, this might look like this:

```
var
   Vector1:      array [1..5] of integer;
   Vector2:      array [1..10] of integer;
   Vector3:      array [7..9] of integer;
      . . .
procedure DefineVector(var Vector: array [Lo..Hi: integer] of integer);
var
  I:             integer;
begin
for I:=Lo to Hi do
  Vector[I]:=I;
end;
      . . .
DefineVector(Vector1);
DefineVector(Vector2);
DefineVector(Vector3);
```

Any of the arrays, regardless of size, can be sent to the procedure `DefineVector`. In passing the array to the procedure, the bounds identifiers ("`Lo`" and "`Hi`") are defined. Inside the procedure, `Lo` and `Hi` can be used anywhere a variable or constant can be used, *except* in declaration statements. That is, you cannot declare another variable such as:

```
var
  NewArray:      array [Lo..Hi] of integer;   { Illegal! }
```

Nor can you "redimension"—change the size of—an array by assigning a value to a bounds identifier:

```
Lo:=3;        { Illegal! }
Hi:=4;        { Illegal! }
```

Nor can you do anything else to try to change such a value; such as pass it by reference to a procedure or function.

Another example of using conformant arrays is in multi-dimensional arrays. As usual in Pascal,

```
array [⟨range1⟩, ⟨range2⟩] of ⟨type⟩
```

is equivalent to

```
array [⟨range1⟩] of array [⟨range2⟩] of ⟨type⟩
```

Suppose you have defined a matrix thus:

```
type
  M4x4=          array [1..4, 1..4] of integer;
var
  M1:            M4x4;
```

You could define a procedure, using conformant arrays, to define the identity matrix:

```
procedure Identity(var Matrix: array [RowMin..RowMax: integer;
                                      ColMin..ColMax: integer] of integer);
var
  Row, Col:      integer;
begin
if Row=61 then Matrix[Row,Col]:=1
          else Matrix[Row,Col]:=0;
end;
```

An additional legality check could be made to ensure that the matrix is square; a non-square identity matrix is a contradiction.

To send multiple conformant arrays to a procedure (or function; all these statements about conformant arrays can be applied to function parameters, too), you just separate them by semicolons in the usual way. Also, you can intermix conformant arrays passed by value and conformant arrays passed by reference[3]:

```
procedure MatMult(    Left:   array[LRowMin..LRowMax: integer;
                                    LColMin..LColMax: integer] of integer;
                      Right:  array[RRowMin..RRowMax: integer;
                                    RColMin..RColMax: integer] of integer;
                  var Answer: array[ARowMin..ARowMax: integer;
                                    AColMin..AColMax: integer] of integer);
var
  Row, Col, Sum:        integer;
  I, J, K:              integer;
begin
if (LColMax-LColMin+1)<>(RRowMax-RRowMin+1) then
  begin
  writeln('For a matrix multiply, the number of columns in the left matrix');
  writeln('must equal the number of rows in the right matrix.  The');
  writeln('matrices passed to the matrix multiply routine failed this');
  writeln('test; resultant matrix zeroed.');
  for Row:=ARowMin to ARowMax do
    for Col:=AColMin to AColMax do
      Answer[Row,Col]:=0;
  end
else
  begin
  for I:=LRowMin to LRowMax do
    for J:=LColMin to LColMax do
      begin
      Sum:=0;
      for K:=LColMin to LColMax do
        Sum:=Sum+Left[I,K]*Right[K,J];
      Answer[I,J]:=Sum;
      end;
  end;
end;
```

[3] If you pass a conformant array to a procedure, and, from that procedure, you wish to pass the array to another procedure, you must pass it (the second time) by reference.

Program Flow 12

Introduction

This chapter contains information on how you can alter the standard sequence of program flow, which is normally one statement after another in sequential order. There are several different areas included in this chapter. They are:

- Standard Pascal branching,
- Procedure and function calls, and
- Procedure variables.

Standard Branching

All of the branching constructs available in standard Pascal are implemented in Workstation Pascal. The following list describes these constructs:

- **if/then/else**
- **for/do**
- **repeat/until**
- **while/do**
- **case/of**
- **goto**

These are described in the *Programming and Problem Solving With Pascal* book.

CASE/OF

HP Pascal supports these two extensions to the standard Pascal **case** statement.

- Subranges for **case** constant lists. For example, if you want the values 4, 5, 6, and 7 to cause the same action, you could type **4..7:** in the case constant list.
- The **otherwise** case. If none of the case constants match the value coming into the **case** statement, the **otherwise** clause, if it exists, will be executed. The **otherwise** clause consists of the word **otherwise** and one statement, which may be compound. Note that there is no colon between the word "**otherwise**" and its statement.

See the *Pascal Language Reference* for more details on these extensions.

Procedures and Functions

HP Pascal incorporates all the standard parameter-passing rules which are in effect for standard Pascal. The following sections document only the HP extensions, and all require the compiler option **$sysprog$** to be in effect.

Relaxed Typechecking of VAR Parameters

The **anyvar** parameter specifier in a function or procedure heading relaxes type compatibility checking when the routine is compiled. This is sometimes useful to allow routines to act on a general class of objects. For instance, an I/O routine may be able to enter or output an array of arbitrary size.

```
$sysprog$              {required}
  . . .
type
  Buffer=              array [0..maxint] of char;
var
  Arr1:                array [2..50] of char;
  Arr2:                array [0..99] of char;
procedure Output_HPIB(anyvar Ary: Buffer; LoBound, HiBound: integer);
⟨procedure body⟩
    . . .
Output_HPIB(Arr1,2,50);
Output_HPIB(Arr2,0,99);
```

Anyvar parameters are passed by reference, not by value; that is, the address of the variable is passed. Within the procedure, the variable is treated as being of the type specified in the heading.

For instance, if an array of 10 elements is passed as an **anyvar** parameter which was declared to be an array of 100 elements, an error may very well occur. The called routine has *no way* of knowing what you actually passed, except perhaps by means of other parameters as in the example above. **Anyvar** should only be used when it's absolutely required, since it defeats the Compiler's normal type-safety rules.

Note

Programs calling routines with **anyvar** parameters should be *very thoroughly debugged!* Careless use of this feature can crash your system.

The above example can be more appropriately implemented using conformant arrays; see the "Data Structures" chapter of this manual.

The ANYPTR Type

Another way to defeat type checking is with the non-standard type **anyptr**. This is a pointer type which is assignment-compatible with all other pointers, just like the constant **nil**. However, variables of type **anyptr** are not bound to a base type, so they *cannot* be de-referenced (i.e., **anyptr_var^** is not permitted). They may only be assigned or compared to other pointers. Passing as a value parameter is a form of assignment.

```
$sysprog$      {required}
  . . .
type
  Pointer1=    ^integer;
  Pointer2=    ^record
                 R1, R2:       real;
               end;
var
  V1,V1a:      Pointer1;
  V2:          Pointer2;
  AnyV:        anyptr;
  Which:       (Type1,Type2);
begin
new(V1);
new(V2);
. . .
if . . . then
  begin
  AnyV:=V1;
  Which:=Type1
  end
else
  begin
  AnyV:=V2;
  Which:=Type2
  end;
. . .
if Which=Type1 then
  begin
  V1a:=AnyV;
  V1a^:=V1a^+1;
  end;
end;
```

The compiler has no way to know if **anyptr** tricks were used to put a value into a normal pointer. If a pointer type which is bound to a small object has its value tricked into a pointer bound to a large object, subsequent assignment statements which dereference the tricked pointer may destroy the contents of adjacent memory locations.

Note

Routines that use the **anyptr** feature should be *very thoroughly debugged!* Careless use of this feature can crash your system.

Procedure Variables and the Standard Procedure CALL

Sometimes it is desirable to store in a variable a "pointer" to a procedure, and then later to call that procedure. For instance, the File System's "Unit Table" is an array which contains (among other things) the location of the driver to be called to perform I/O on each logical unit.

A variable of this sort is called a "procedure variable," or "procvar," for short. The "type" of a procedure variable is a description of the parameter list it requires. That is, a procedure variable can be bound to a particular procedure, which must have a particular type of heading.

```
$sysprog$    {REQUIRED for procvars and CALL}
  . . .
type
  ProcVar=          procedure(Op: integer);
var
  P:                ProcVar;
  I:                integer;
procedure Q(Op: integer);    {identically structured parameter list}
  . . .
P:=Q;                 {P gets the address of Q; in effect, P points to Q}
call(P,I); {execute Q}{name of procvar, then appropriate parameter list}
```

A procedure variable is invoked by the standard procedure `call`, which takes the procedure variable as its first parameter, and a further list of parameters just as they would normally be passed to the procedure having the corresponding specification.

It is not possible to create a "function variable", that is, a variable which can hold the address of a function.

Don't assign an inner (non-global) procedure to a procedure variable which isn't declared in the same block as the procedure being assigned. Such a variable might be called later, after exiting the scope in which the procedure was declared. The stack (local variables, etc.) assumed by the procedure will have been released, giving unpredictable operation, possibly fatal to the system.

Numeric Computation 13

Introduction

When people think about computers, the first thing that they often think of is number-crunching, the giant calculator with a brain. Whether this is an accurate impression or not, numeric computations are an important part of computer programming.

Numeric computations deal exclusively with numeric values. Thus, adding two numbers or finding a sine or a logarithm are all numeric operations, while converting a number to a string or a string to a number are generally not. (See page 13-12 for some examples.) (Converting numbers to strings and strings to numbers is covered in the "String Manipulation" chapter.)

The most fundamental numeric operation is the assignment operation, achieved with the ":=" assignment operator. Thus the following statements are assignment statements:

```
A:=1;
Sine := sin(Theta);
X:=X+1;
```

Numeric Data Types

There are two numeric data types in Pascal, INTEGER and REAL. The valid range for REAL numbers is approximately:

$$-1.797\,073\,134\,862\,315\times10^{308} \text{ through } 1.797\,073\,134\,862\,315\times10^{308}$$

The smallest non-zero REAL value allowed is approximately:

$$\pm2.225\,073\,858\,507\,202\times10^{-308}$$

A REAL can also have the value of zero.

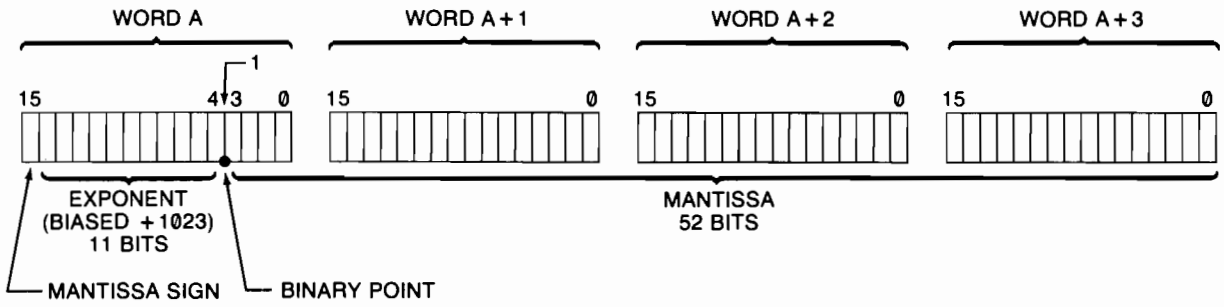
An INTEGER can have any whole-number value from:

$$-2\,147\,483\,648 \text{ through } +2\,147\,483\,647$$

Note

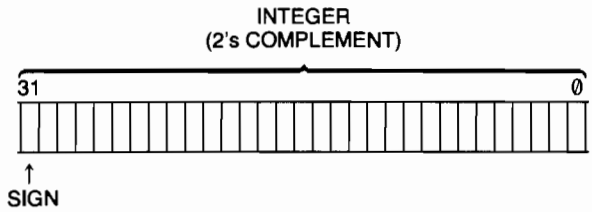
These ranges are implementation dependent. Other Pascal implementations may have different values.

Internal Numeric Formats



$$-1^{\text{mantissa sign}} \times 2^{\text{exponent} - 1023} \times 1.\text{mantissa}$$

Storage Format for REAL Variables



Storage Format for INTEGER Variables

Note

These formats are hardware dependent and operating system dependent. Other computers which support Pascal may have very different internal formats.

Declarations

In Pascal, you must declare all variables before using them, and both INTEGER and REAL data types are provided for declaring numeric variables:

```
var
  I, J:      integer;
  Days:      array [1..5] of integer;
  Weeks:     array [1..5] of array [1..17] of integer;
  X, Y:      real;
  Voltage:   array [1..4] of real;
  Hours:     array [1..5, 8..13] of real;
```

The above statements declare, both for integers and reals, the following:

- Two scalars,
- A one-dimensional array, and
- A two-dimensional array.

A scalar is a variable which can, at any given time, represent a single value. An array is a subscripted variable, and can contain multiple values, accessed by subscripts. You must specify both the lower and upper bounds of an array. Details on declarations of arrays and how to use them are provided in the "Data Types" chapter of this manual.

Type Conversions

The computer will automatically convert integers to real numbers in assignment statements and when parameters are passed by value in function and procedure calls. When parameters are passed by reference the conversion will not be made and a type-mismatch error will be reported. The computer will *not* automatically convert a real number to an integer; you must explicitly tell the computer to do it, and how to do it. There are two ways to convert a real number to an integer:

- The **round** function, which rounds the real value to the closest integer ($n.5$ rounds up to $n+1$), and
- The **trunc** function, which truncates the real value to the next integer toward 0.

For both of these functions, the sign (positive or negative) is not taken into account during the operation. You could think of them as doing these three operations:

1. Take the absolute value of the argument,
2. Do the operation,
3. Re-attach the original sign to the result.

An example of where this is significant is in the **trunc** function. It rounds toward 0, not toward $-\infty$. That is, `trunc(1.7)` yields 1, as expected, but `trunc(-1.7)` yields -1, not -2. It very literally truncates, it does not round to the next integer less than or equal to the argument.

Whenever numbers are converted from REAL to INTEGER representations, information can be lost. There are two potential problem areas in this conversion: rounding errors and range errors.

The computer may automatically convert between types when an assignment is made, and this presents no problem when an INTEGER is converted to a REAL. However, when you convert a REAL to an INTEGER, the REAL is modified (in whatever way you specified) to the closest INTEGER value. When this is done, all information about the value to the right of the radix (decimal point) is lost. If the fractional information is truly not needed, there is no problem, but converting back to a REAL will not reconstruct the lost information—it stays lost.

Another potential problem with REAL to INTEGER conversions is the difference in ranges. While REAL values range from approximately -10^{308} to $+10^{308}$, the INTEGER range is only from **minint** through **maxint**, or −2 147 483 648 through +2 147 483 647 (approximately -10^9 thru $+10^9$). Obviously, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate integer overflow errors.

While the rounding problem is important, it does not generate an execution error. The range problem *can* generate an execution error, and you should protect yourself from crashing the program by either testing values before assignments are made, or by using **try/recover** to trap the error[1], and making corrections after the fact.

The following fragment shows a method to protect against INTEGER overflow errors, although be aware that this method imposes minimum and maximum limits on the value[2]:

```
if X<minint then X:=minint
else
  if X>maxint then X:=maxint
  else
    X:=trunc {or round} (X);
```

Both these methods limit the excursion, but lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

```
if (X<minint) or (X>maxint) then
  OutOfRange:=true
else
  X:=round {or trunc} (X);
```

[1] See the chapter on "Error Trapping and Simulation" for details on error recovery.
[2] Later in this chapter, a method which truncates numbers outside the **minint..maxint** range is shown.

Precision and Accuracy: The Machine Limits

Your computer stores all REAL variables with a sign, approximately 15 significant digits, and the exponent value. For most applications, this resolution is well beyond actual program needs. However, when high-resolution numerical analysis requires accuracy approaching the limits of the computer, round-off errors must be considered.

For many engineering and other applications, rounding errors are not a problem because the resolution of the computer is well beyond the limitations of most scientific measuring devices.

Rounding errors should be considered when conversions are made between decimal digits and binary form. Input/output operations are one time when this occurs. Given the format used for REALs, the conversion REAL→decimal→REAL will yield an identity only if the REAL→decimal conversion produces a 17-decimal-digit mantissa and the calculations for the conversions are done in extra precision. This is not the case on the Workstation Pascal. Therefore, several things can be said about these conversions:

- Up to and including 16 decimal digits are allowed when storing a number in internal form. If there are more digits, they are ignored.
- Up to and including 15 decimal digits may be output when converting a REAL for printing, display, etc. A full 16-digit conversion is not allowed because there are not 16 full digits of precision.
- It is possible for two distinct decimal numbers to map onto the same REAL number because the binary mantissa does not have enough bits to represent all 16 decimal digits. This can happen only if the decimal numbers are specified to 16-digits.
- It is possible for two distinct REAL numbers to convert to the same decimal number even if the conversion is done to 15-decimal-digit accuracy. Therefore, you cannot use a comparison of the digits in printed or displayed numbers to check for equality.
- All distinct 15 digit decimal strings have a correct distinct REAL representation, but it is not always possible to map them onto their correct representation because REAL multiplies are not done in extra precision, and the table entries are only 64 bits. In other words, the decimal→REAL conversion may produce a REAL that differs from the true representation by a maximum of two bits.

There are references at the end of this chapter to documents that contain further information on the subject of representing real numbers.

Evaluating Scalar Expressions

The Hierarchy

If you look at the expression 2+4/2+6, it can be interpreted several ways:

- 2+(4/2)+6 = 10
- (2+4)/2+6 = 9
- 2+4/(2+6) = 2.5
- (2+4)/(2+6) = .75

Computers do not deal well with ambiguity, so an arbitrary hierarchy is used for evaluating expressions to eliminate any questions about the meaning of an expression. When the computer encounters a mathematical expression, an expression evaluator is called. If you do not understand the expression evaluator, you can easily be surprised by the value returned for a given expression. In order to understand the expression evaluator, it is necessary to understand the valid elements in an expression and the evaluation hierarchy (the order of evaluation of the elements).

Six items can appear in a numeric expression; operators, constants, variables, intrinsic functions, user-defined functions and parenthesis. Operators modify other elements of the expression. Constants and variables represent numeric values in the system. Functions, both intrinsic and user-defined, return a value which replaces them in the evaluation of the expression. Parenthesis are used to modify the evaluation hierarchy.

The following table defines the hierarchy used by the computer in evaluating numeric expressions.

Math Hierarchy

Precedence	Operator
Highest	Parentheses; they may be used to force any order of operation
	Functions, both user-defined and machine-resident
	Multiplication and division: `*`, `/`, `mod`, `div`.
	Addition, subtraction, monadic plus and minus: `+`, `-`.
Lowest	Relational and Boolean operators: `=`, `<>`, `<`, `>`, `<=`, `>=`, `NOT`, `AND`, `OR`.

The boolean operators, `NOT`, `AND`, and `OR`, are included because of their utility in creating step functions (see the section "Step Functions" later in this chapter).

When an expression is being evaluated, it is read from left to right and operations are performed as encountered, unless a higher precedence operation is encountered immediately to the right of the operation encountered, or unless the hierarchy is modified by parentheses. If the computer cannot deal immediately with the operation, it is stacked, and the evaluator continues to read until it encounters an operation it can perform. It is easier to understand if you see how an expression is actually handled. The following expression is complex enough to demonstrate most of what goes on in expression evaluation.

```
A:=5+3*(4+2)/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0));
```

In order to evaluate this expression, it is necessary to have some historical data. Since trigonometric functions in Pascal deal only with radians, we will assume that X=$\pi/2$, and that the user-defined function **Neg1** returns -1. Evaluation proceeds as follows:

```
5+3*(4+2)/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
  5+3*6/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
    5+18/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
      5+18/1+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
       5+18+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
        23+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
      23+X*ord(false)+Neg1*ord((X<1.5) and (X>0.0))
          23+X*0+Neg1*ord((X<1.5) and (X>0.0))
           23+0+Neg1*ord((X<1.5) and (X>0.0))
            23+Neg1*ord((X<1.5) and (X>0.0))
            23+-1*ord((X<1.5) and (X>0.0))
             23+-1*ord(false and (X>0.0))
              23+-1*ord(false and true)
                  23+-1*ord(false)
                     23+-1*0
                      23+0
                       23
```

The Delayed Binding Surprise

The computer delays binding of a variable to its value as long as possible. In the actual evaluation, a pointer to the location of a variable is what is stacked. This means that if a variable exists in an area of memory accessible to both the main program and a user-defined function, is used in an expression that also calls the user-defined function, and is modified in the function, the value of the expression can be surprising, although not unpredictable. For example, if we define a function **Neg1** that returns a negative 1, we would expect the following lines to print 2.

```
X:=3;
Y:=X+Neg1;
writeln(Y);
```

However, if the these lines are in the following environment:

```
program DBinding(input, output);
var
  X, Y:          integer;

function Neg1: integer;
begin
X:=500;
Neg1:=-1;
end;

begin
X:=3;
Y:=X+Neg1;
writeln(Y);
end.
```

The actual result will be 499—surprising, but not unpredictable. The same thing will happen if the variable is passed by reference and modified in the user-defined function. Therefore, be careful when you use a user-defined function to modify values of global variables. They are designed for returning a single value, and are best reserved for that.

Operators

There are two types of numeric operators in Pascal: monadic and dyadic:

- A **monadic** operator performs its operation on the expression immediately to its right; e.g., **+**, **-**, **not**.
- A **dyadic** operator performs its operation on the two values it is between; e.g., **^**, *****, **/**, **mod**, **div**, **+**, **-**, **=**, **<>**, **<**, **>**, **<=**, **>=**, **and**, **or**.

  A **comparison** operator returns **true** or **false**, based on the result of a relational test of the operands it separates. The comparison operators are a subset of the dyadic operators that produce *boolean* results; e.g., **<**, **>**, **<=**, **>=**, **=**, **<>**.

While the use of most operators is obvious from the descriptions in the language reference, some of the operators have uses and side-effects that are not always apparent.

Expressions, Calls, and Functions

Numeric expressions can be passed by value to procedures and functions, if the corresponding formal parameter does not have the keyword **var** before the parameter name (assume for the moment it does not). Thus **5+X** is obviously passed by value. Not quite so obviously, **+X** is also passed by value. The monadic operator makes it an expression.

Step Functions

The comparison operators are obviously useful for conditional branching (**IF/THEN** statements), but are also valuable for creating numeric expressions representing step-functions. For example, let's try to represent the function:

- if **Select** < 0
  then **Result** $= 0$
- if $0 <=$ **Select** < 1
  then **Result** equals the square root of $A^2 + B^2$.
- if **Select** $>= 1$ (any other value)
  then **Result** $= 15$

It is possible to generate the required response through a series of **IF/THEN** statements, but it can also be done with the following expression:

```
Result:=0                  *ord(Select<0)+
        sqrt(sqr(A)+sqr(B))*ord((Select>=0) and (Select<1))+
        15                 *ord(Select>=1);
```

While the technique may not please the purist, it actually represents the step function very well. The boolean expressions cause the **ord** function to return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0, and are not included in the result. The value assigned to **Select** before the expression is evaluated determines the computation placed in the result.

This technique can be used to represent a cyclicity as well; every time through a particular set of statements, the "next" of a list of variables is selected. At the end of the list, the list is repeated from the beginning, ad infinitum. Boolean expressions, constants, and variables can be included in numeric expressions if they are "converted" to numeric by the intrinsic function **ord**. What we haven't seen yet is that the boolean expressions can be generated by comparing *anything* that is comparable; e.g., numbers, characters, strings, etc. Note in the following examples, **X** and *a*, *b*, *c*, and *d* can be any of the type mentioned (numbers, characters, strings), as long as they are *all* of the same type. For example:

`X:=ord(X=0);` This expression alternates the value of **X** between 0 and 1.

`X:=a*ord(X=b)+b*ord(X=a);` This expression alternates between the arbitrary values of *a* and *b*, as long as $a \neq b$ (x must initially be either *a* or *b*). Note that this algorithm can be extended to cycle through any number of values, as the next example shows.

```
X:=a*ord(X=d)+
   b*ord(X=a)+
   c*ord(X=b)+
   d*ord(X=c);
```

This expression cycles through the four values *a*, *b*, *c*, and *d*. Make sure $a \neq b \neq c \neq d \neq \ldots$, because if any value equals any other value, the process will loop without completing the series. Also, *X* must start out containing on of *a*, *b*, *c*, or *d*. As mentioned, this algorithm can be extended to any number of values, but it quickly gets cumbersome. The algorithm fills the need best when the values being cycled through exhibit no discernible pattern.

A permutation array is a more portable structure for doing the same thing, and is easily extendible. For the above, given *a*, *b*, *c*, *d* as constants:

```
type
  perm4 = array[0..3] of integer;
const
  cycleperm = perm4[a,b,c,d]; {could be variable}
  ...
var
  permindex : integer;
  ...
begin
  permindex := 0;  {select "a" to start}
  ...
  x := cycleperm[permindex mod 4];
  permindex := permindex +1 {select next}
  ...
```

Again: if, as in the above examples, `X` is a number, you could cycle through 14, 23, 4, and −45. If, in the above examples, `X` is a character, you could cycle through `'x'`, `'&'`, `'A'`, and `'f'`. However, you cannot "multiply" a character by a number, but you can convert between characters and numbers easily. For example, assume `A`, `B`, `C`, `D` and `X` are of type `char`:

```
A:='x';
B:='&';
C:='A';
D:='f';
X:=A;        {or B or C or D...}
  . . .
X:=chr(ord(A)*ord(X=D)+
       ord(B)*ord(X=A)+
       ord(C)*ord(X=B)+
       ord(D)*ord(X=C));
```

If you want to cycle through strings, for example, `'Artichoke'`, `'I bought a centipede'`, `'quark'`, and `'Oh.'`, you could use the function `strrpt` for to do the "multiplication" (see the "String Manipulation" chapter for more on `strrpt`). For example (assume `A`, `B`, `C`, `D` and `X` are strings):

```
A:='Artichoke';
B:='I bought a centipede';
C:='quark';
D:='Oh.';
X:=A;        {or B or C or D...}
  . . .
X:=strrpt(A,ord(X=D))+
   strrpt(B,ord(X=A))+
```

```
strrpt(C,ord(X=B))+
strrpt(D,ord(X=C));
```

If you want to cycle through enumerated-type values, there is no easy way to do it other than putting the values in an array, and cycling through the subscripts.

Be aware that use of **ord** often indicates non-portable code, especially when applied to ordering of character sets.

Making Comparisons Work

If you are comparing integers, no special precautions are necessary. However, if you are comparing real values, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding and other limits inherent in the system. For example, consider the use of comparison operators in **if/then** statements to check for equality in any situation resembling the following:

```
A:=2.53765477;
if sin(A)^2+cos(A)^2=1.0 then
  writeln('Equal')
else
  writeln('Not Equal');
```

You may find that the equality test fails due to rounding errors or other errors caused by the inherent limitations of finite machines. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

A good example of equality error occurs when multiplying or dividing data values. A product of two non-integer values nearly always results in more digits beyond the decimal point than exists in either of the two numbers being multiplied. Any tests for equality must consider the *exact* variable value to its greatest resolution. If you cannot guarantee that all digits beyond the required resolution are zero, there are two techniques that can be used to eliminate equality errors:

- Use the absolute value of the difference between the two values, and test for the difference less than a specified limit. Here is an example of the absolute value method of testing equality. In this case, a difference of less than 0.001 is assumed to be evidence of adequate equality.

```
if abs(C-F)<0.001 then
  writeln('C is equal to F within 0.001')
else
  writeln('C is not equal to F within 0.001');
```

- Use the absolute value of the *relative* difference between two values, and test for the difference less than a specified limit:

```
if abs((C-F)/C)<10E-8 then
  writeln('Relative difference between C and F less than 10E-8')
else
  writeln('Relative difference between C and F greater than 10E-8');
```

This technique has the advantage that no additional statements are invested in overhead while preparing the data for evaluation. It also enables you to easily establish tolerance limits in making value comparisons, a capability that is useful in production and testing applications.

Numerical Functions

The resident functions are the functions that are part of the Pascal language (also called intrinsic). The following functions are available:

Function	Description
`abs`	Returns the absolute value of an expression.
`arctan`	Returns the arctangent of an expression.
`binary`	Takes a string consisting of '1's and '0's and converts it to an integer.
`cos`	Returns the cosine of the angle represented by the expression.
`exp`	Raise the Napierian e to a power. $e \approx 2.718\,281\,828\,459\,05$.
`hex`	Takes a string of hexadecimal digits ('0' thru '9' and 'A' thru 'F') and converts it to an integer.
`ln`	Returns the natural logarithm (Napierian base e) of an expression.
`octal`	Takes a string of octal digits ('0' thru '7') and converts it to an integer.
`odd`	Takes an integer argument and returns a *boolean* result: `true` if and only if the absolute value of the integer argument is odd.
`round`	Returns the closest integer to the real argument. The result is of type `INTEGER`. `round(X) = sign(X)*trunc(abs(X)+0.5)` where `sign(X) = ord(X>0.0) - ord(X<0.0)`.
`sin`	Returns the sine of the angle represented by an expression.
`sqr`	Returns the square of an expression.
`sqrt`	Returns the square root of an expression.
`trunc`	Returns the integer part of the real argument; the fractional part is removed. The result is of type `INTEGER`.

Also, we should mention `div` and `mod`, although they are operators and not functions. The `div` operator does an integer divide; that is, it does a division and discards any fractional part. The `mod` operator returns the remainder of an integer division. `Z:=X mod Y` is equivalent to:

```
Z:=X-Y*(X div Y);
if Z<0 then Z:=Z+Y;
```

Dealing with Angles and Such

Before we get into the functions that deal with angles, let's discuss the angles themselves.

The Units

In Pascal, angles are always considered to be in radians, but people often want to deal with angles in degrees, or grads. Here are definitions of these units of angular measure:

- **Radians:** A radian is the unit of angular measure subtended by a piece of circumference of a circle whose length is equal to the radius of the circle. That is, if you take a line whose length is the radius of a circle, and bend that line around the circumference, the angle subtended by the bent radius is one radian.
- **Degrees:** A degree is $\frac{1}{90}$ of a right angle. Thus, there are 360 degrees in a complete revolution.
- **Grads:** A grad is 1% of a right angle. Thus, there are 400 grads in a complete revolution.

In a nutshell, these are the relationships between these units of angular measure:

$$
\begin{array}{rcrcl}
\text{radians} & = & \text{degrees} \times \frac{\pi}{180} & \approx & \text{degrees} \times 0.017\,453\,292\,519\,943\,3 \\
\text{radians} & = & \text{grads} \times \frac{\pi}{200} & \approx & \text{grads} \times 0.015\,707\,963\,267\,949\,0 \\
\text{degrees} & = & \text{radians} \times \frac{180}{\pi} & \approx & \text{radians} \times 57.295\,779\,513\,082\,3 \\
\text{degrees} & = & \text{grads} \times \frac{9}{10} & = & \text{grads} \times 0.9 \\
\text{grads} & = & \text{radians} \times \frac{200}{\pi} & \approx & \text{radians} \times 63.661\,977\,236\,758\,1 \\
\text{grads} & = & \text{degrees} \times \frac{10}{9} & \approx & \text{degrees} \times 1.111\,111\,111\,111\,111\,1
\end{array}
$$

The Functions

There are three functions which HP Pascal provides for dealing with functions: **sin**, **cos**, and **arctan**. However, since the default mode for all angular measure is radians, there needs to be a conversion from the units used in the program to radians. Assume, for example, that the program is considering all angles to be in degrees, and that the variable **Theta** holds the angle. The sine of **Theta** (which is in degrees) is:

```
Sine:=sin(Theta*0.01745329252);
```

When going the other direction, divide by the constant rather than multiply[3]. For example, say we want to calculate the arctangent of **X**, and have the answer in degrees:

```
Theta:=arctan(X)/0.01745329252;
```

Pascal traditionally is somewhat weak in the area of numerical functions, and HP Pascal is no different. A person can do hardly any trigonometric calculations from only the three functions provided, unless he knows some of the trigonometric relationships with which to derive the other needed functions. It is beyond the scope of this manual to provide very high speed algorithms to directly calculate the other trig functions; see a math book for those. However, it is within the scope of this manual to provide equations with which you can derive the appropriate functions from combinations of others. In the equations that follow, radians are assumed. If you wish to work in other units of angular measure apply the formulae above to convert to and from the desired unit of measure. Here are some of the trigonometric relationships.

[3] Speed can be increased by multiplying by the reciprocal of the degree-to-radian number, rather than dividing by it. The computer does a multiply by 57.295 779 51 faster than a divide by 0.017 453 292 52.

Perhaps the best-known of the trig relationships is the following:

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

This shows that to calculate the tangent of an angle; take the sine of that angle and divide it by the cosine of that same angle. Note, however, that for odd multiples of $\pi/2$ (90°) the cosine is zero, so you must take appropriate measures to avoid dividing by zero in these cases.

A little less well-known are the arcsine and arccosine identities: The arcsine is defined thus:

$$\arcsin x = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

In Pascal:

```
Theta:=arctan(X/sqrt(1-sqr(X)));
```

Here also, there is some danger of dividing by zero. When $x=1$, the denominator evaluates to zero. This divide by zero can be avoided in either of two ways. You could test for $x=1$ and branch to a separate place to give the value, or a less cluttered way is the following:

$$\arcsin x \approx \arctan\left(\frac{x}{\sqrt{1-x^2+\epsilon}}\right)$$

where ϵ is some very small number e.g., 10^{-100}. Then, when $x=1$, the denominator will not evaluate to zero, but a small number. The whole expression then evaluates to a very large (for all practical purposes, infinite) number. This is consistent with the desired behavior of the expression, because the argument for the arctangent function can range from approximately -10^{308} to $+10^{308}$.

Similar to the arcsine function is the arccosine function:

$$\arccos x = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$$

But again, the divide-by-zero threat still exists; this time, when $x=0$. The resolution of this problem is similar to that of the arcsine function above:

$$\arccos x = \arctan\left(\frac{\sqrt{1-x^2}}{x+\epsilon}\right)$$

or

```
Theta:=arctan(sqrt(1-sqr(X))/(X+Eps))
```

The reciprocals of the three main trig functions are the secant, cosecant, and cotangent:

$$\sec\theta = \frac{1}{\cos\theta}$$

$$\csc\theta = \frac{1}{\sin\theta}$$

$$\cot\theta = \frac{1}{\tan\theta}$$

This gives us the main three trigonometric functions **sin**, **cos**, and **tan**; their inverses **arcsin**, **arccos**, and **arctan**; and their reciprocals **sec**, **csc**, and **cot**.

The arctangent function supplied by the Pascal language is fine for many applications, but for others, it doesn't show you enough information. For example, say you have a point which has the Cartesian coordinates (3,4) and you want to determine its angle from the origin. This is quite simple: evaluate the arctangent of (Δy/Δx), or `arctan(4/3)`; it comes to about 0.93 radians, or about 53.13 degrees. This is the correct answer.

But here's the problem. Suppose that the point was (−3,−4). Calculating arctan(Δy/Δx), or `arctan(-4/-3)` still results in 0.93 radians, or about 53 degrees, but this answer is off by 180 degrees! The problem arises from the fact that a negative number divided by a negative number gives the same answer as a positive number divided by a positive number.

Another problem arises when the point of interest lies on the Y-axis. This means x is zero, and again we're faced with a divide-by-zero problem.

The resolution of this requires that we pass both x and y to the new arctangent function; do not do the division beforehand, because then the signs are lost.

Let's look at every possibility of x and y. They both can be negative, zero, or positive.

$$\arctan(y,x) = \begin{cases} \arctan(y/x) + \pi & \text{if } x < 0 \text{ and } y < 0; \\ \arctan(y/x) + \pi \ (= \pi) & \text{if } x < 0 \text{ and } y = 0; \\ \arctan(y/x) + \pi & \text{if } x < 0 \text{ and } y > 0; \\ \frac{3}{2}\pi & \text{if } x = 0 \text{ and } y < 0; \\ 0 \text{ (by definition)} & \text{if } x = 0 \text{ and } y = 0; \\ \frac{1}{2}\pi & \text{if } x = 0 \text{ and } y > 0; \\ \arctan(y/x) & \text{if } x > 0 \text{ and } y < 0; \\ \arctan(y/x) \ (= 0) & \text{if } x > 0 \text{ and } y = 0; \\ \arctan(y/x) & \text{if } x > 0 \text{ and } y > 0; \end{cases}$$

As you can see, there is a pattern that emerges. If x>0, the normal arctangent function does well. If x<0, the normal arctangent function is consistently off by one-half revolution; we could just add π radians (180°) to the result. If x=0, we need to check the sign of y and deal with it accordingly. If x and y are both zero, the arctangent is undefined; you're asking the computer to calculate the direction of a point. But, to keep the computer happy, let's define (somewhat arbitrarily) the result to be 0.

Taking the above information and translating it into Pascal, you come up with an arctangent function that goes something like the following (note that we're using one of the tricks we learned in the "Step Functions" section):

```
const
  Pi=           3.1415926535897932384;
      .
      .
      .
if X=0.0 then
  Atan:=(Pi/2
         +Pi*ord(Y<0.0))
            *ord(Y<>0.0)
else
  Atan:=arctan(Y/X)
        +Pi*ord(X<0.0)
        +2*Pi*ord((X>0.0) and (Y<0.0));
```

The "then" clause of the `if` statement says this:

1. We've already determined that $x=0$; thus, the point is on the Y-axis. Therefore, assume $\pi/2$, or 90° (straight up).
2. Now, if $y<0$, add π (180°) to make the angle straight down.
3. One final check: if x and y are both zero, return zero.

The "else" clause of the `if` statement says this:

1. Take the arctangent of y/x.
2. If $x<0$, add π (180°).
3. If $x>0$ and $y<0$ (if the point is in the fourth quadrant), add 2π (360°). This ensures that the result ranges from 0 to 2π, rather than $-\frac{\pi}{4}$ to $+\frac{3\pi}{4}$.

Another class of trigonometric operations is the hyperbolic functions. Although we won't go into detail on how to use them, they are provided for your reference here.

$$\sinh z = \frac{e^z - e^{-z}}{2}$$

$$\sinh^{-1} z = \ln(z + \sqrt{z^2 + 1})$$

$$\operatorname{csch} z = \frac{1}{\sinh z}$$

$$\cosh z = \frac{e^z + e^{-z}}{2}$$

$$\cosh^{-1} z = \ln(z + \sqrt{z^2 - 1})$$

$$\operatorname{sech} z = \frac{1}{\cosh z}$$

$$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh^{-1} z = \frac{1}{2} \ln\left(\frac{1+z}{1-z}\right)$$

$$\coth z = \frac{1}{\tanh z}$$

Range Limits

It is sometimes necessary to limit the range of excursion of a variable (as in the discussion of REAL to INTEGER conversions mentioned in the introduction to this chapter). It is possible to do this with **if/then** statements:

```
if X>Maxx then X:=Maxx;
if X<Minx then X:=Minx;
```

It is more convenient to use **max** and **min** functions which can be defined thus:

```
function min(X, Y: real): real;
begin                                   {function "min"}
if X<Y then min:=X
else min:=Y;
end;                                    {function "min"}

function max(X, Y: real): real;
begin                                   {function "max"}
if X>Y then max:=X
else max:=Y;
end;                                    {function "max"}
```

For example:

```
X:=min(max(X,Minx),Maxx)
```

Note that **max** is used to establish the lower bound, and **min** is used to establish the upper bound. If you think about it a minute, it makes sense.

Rounding

Rounding occurs frequently in computer operations. The most common rounding occurs in printouts and displays, where it can be handled effectively with the formatting numbers (the numbers after the colons) in the output operation.

Sometimes it is necessary to round a number in a calculation, to eliminate unwanted resolution. There are three basic types of rounding:

- Rounding to a number of decimal places (limiting fractional information);
- Rounding up, down or to the nearest x, where x is any number, real, or integer, except zero; and
- Rounding to a total number of significant digits.

All three types of rounding have their own applications in programming.

Rounding to a Number of Decimal Places

The first, and most basic form of rounding is a special case of the first method above—that of rounding to a number of decimal places—but rounding always takes place to the nearest 10^0, or 1. The function to do this is called **round**, and it is part of the Pascal language. It was covered earlier in this chapter, with a reference to a subsequent function with which you could round real numbers *outside* the range of the integers. That function comes here.

In the previous section, an algorithm for truncating real numbers outside the range of **minint** to **maxint** was discussed. Using this same algorithm, rounding is only trivially more involved.

Rounding to the nearest integer (to the nearest 10^0) is merely a matter of truncating after adding 0.5. Imagine rounding 3.2 to the nearest integer; it rounds to 3. If we add 0.5 to 3.2, and then truncate, we again get 3. Imagine rounding 3.8 to the nearest integer; it rounds to 4. If we add 0.5 to 3.8, and then truncate, we get 4.

The only deviation from this algorithm is for negative numbers. What is often done is that the number is made positive, the rounding operation is done, and then the original sign is re-applied. For example, for rounding −3.2, you would note that it is negative, do the rounding operation on the absolute value of the argument, and re-apply the original sign to the result.

Note

In the rounding examples that follow, the range of numbers to be rounded is assumed to be in the **minint**..**maxint** range; thus, the standard Pascal function **round** will be used. If the numbers to be rounded are outside of this range, use the same algorithms stated, but do the rounding by adding 0.5 and then truncating with the "big number truncator" mentioned in the previous section.

The next logical step is to allow rounding to any power of ten, not just 10^0, as above. The idea is to eliminate decimal representation beyond a specific power of ten. A simple approach to it is to push the desired decimal information to the left of the radix, use **round** to get rid of the undesired decimal information, then reposition the radix correctly.

What must be done is this:

- Divide the number by the appropriate power of 10 to move the digit which will be the rightmost significant digit to just left of the decimal point.
- Round to 10^0, as usual.
- Multiply by the same appropriate power of ten to put the number back into the original order of magnitude.

For example, suppose you want to round 3.14159265 to the nearest 10^{-2}, or hundredth. First, you divide it by 10^{-2}, to get 314.159265. Next, round in the usual way, resulting in 314. Finally, multiply by 10^{-2}, to return the number to the original order of magnitude: 3.14.

Rounding to the Nearest X

All rounding applications don't fit nicely into the "power of 10" pattern mentioned above. What if you wanted to round to the nearest 25? Or 37? Or 0.123 or ¼? Some applications require rounding to the nearest multiple of some pretty unusual numbers.

This, again, is a logical extension of the previous method where we were dealing with powers of 10. Say, for example, that we want to round 19.2 to the nearest dozen. The method is simply:

```
Rounded:=round(19.2/12)*12;
```

Or, more generically, if `N` is the number to be rounded, and `M` is the number to be rounded to:

```
Rounded:=round(N/M)*M;
```

Rounding to N Significant Digits

There is a tendency for the number of decimal places to grow as calculations are performed on the results of other calculations. One of the first things covered in training for engineering and the sciences is how to handle the growth of the number of decimal places in a calculation. If the initial measurements from an experiment produced three digits of information per reading, it is very misleading to produce a seven-digit number as the result of a long series of calculations. Rounding to a specific number of significant digits allows you to eliminate the unwanted digits, to produce more realistic calculations and answers.

The following algorithm is portable among most Pascal implementations. In the process of rounding to a certain number of significant digits, you must address the fact that you don't know where the decimal point is going to be, and the algorithm shouldn't care anyway. Taking this factor into account requires one more step than the previously mentioned rounding methods. The step is: Find out how far the decimal point has to be moved in order to position the number such that a regular rounding operation can be done. In all, the steps are as follows. Assume `X` is the number to be rounded, and `Digits` is the number of significant digits the result is to exhibit.

1. To find out how far the number is to be shifted, make a number which is the next larger order of magnitude; i.e., 1×10^n. This is accomplished by taking the logarithm[4] to the base 10, rounding it *up* to the next integer, and taking 10 to that power.
2. Shift the number again by dividing it by an appropriate power of 10 in order to take into account the number of digits to which to round.

3. Round in the usual way.
4. Shift back the number of digits found in Step 2.
5. Shift back the number of digits found in Step 1.

Implementing this in Pascal is not too difficult. Following is a section of code which would go into a function named **DRound**, which rounds to a certain number of digits. There are several ways to combine steps in this code segment to increase speed, but they were left out to maintain readability. Assume the functions **TenToThe** and **Log10** exist and calculate a power of ten, and the common log, respectively.

```
var
  DigitPower, MagnitudePower:   real;
      .
      .
      .
if Digits>=15 then
  DRound:=X
else
  if Digits<=0 then
    DRound:=X
  else
    begin
    MagnitudePower:=TenToThe(trunc(Log10(abs(X)))+1);
    DigitPower:=TenToThe(-Digits);
    X:=X/MagnitudePower;
    X:=round(X/DigitPower)*DigitPower;
    X:=X*MagnitudePower;
    DRound:=X;
    end;
```

Logarithms and Powers

There are two functions resident in Pascal which deal with logarithms: **ln**, which takes the natural log[5], and **exp**, which takes the natural antilog, or takes e to a power. With these two functions, we can do quite a bit.

X to the Yth Power

One of the logarithmic identities is

$$x^y = b^{y \log_b x}$$

where b is any non-zero number base. Since this works with any number base, e will work nicely:

$$x^y = e^{y \ln x}.$$

Knowing this, it is straightforward to take any number to any power. For example, to find out how many cubic feet in a cubic mile, you take 5280^3, or

```
CubicFeet:=exp(3*ln(5280));
```

When x, above, is a commonly used number, you can save computer time by calculating the natural log of it once, and then hard-coding it. The increase in speed can be significant, because both **exp** and **ln** are complex, relatively slow functions to calculate.

[4] Assume for the moment that functions exist whereby the common logarithm ($\log_{10} x$) common antilogarithm (10^x) can be obtained. The next section in this chapter illustrates how to implement these in Pascal.

[5] The "natural logarithm" is a logarithm based on the Napierian number e, which equals approximately 2.718 281 828 459 045.

For example, to calculate 10 to a power, you could execute this statement every time:

```
Y:=exp(X*ln(10));
```

However, `ln(10)` is not going to change, so speed can be increased by converting this to an equivalent value: 2.302 585 092 994 05.

```
Y:=exp(X*2.30258509299405);
```

This approach can be taken with any number base.

The Xth Root of Y

Another logarithmic identity comes into play here:

$$\sqrt[x]{y} = y^{1/x}$$

This says that the xth root of y is obtained merely by taking y to the power of the reciprocal of x. After taking the reciprocal—just dividing the number into 1—use the approach immediately above for taking a number to a power.

In Pascal, to take the cube root of 27, it would be:

```
Y:=exp(1/3*ln(27));
```

Log to Any Base

So far, we have been looking at logarithms to the base e exclusively, with a minor excursion into base 10. But logarithms exist in any base, so how can you figure a log to any user-specified base? The following derivation illustrates what can be done.

The definition of logarithms:	$\log_b x = y$
$\therefore$ (apply a logarithmic identity)	$b^y = x$
$\therefore$ (take natural log of both sides)	$\ln b^y = \ln x$
$\therefore$ (apply logarithmic identity to left side)	$y \ln b = \ln x$
$\therefore$ (divide both sides by $\ln b$)	$y = \ln x / \ln b$

What this means is that we can calculate the logarithm *to any base* of a number by dividing the log of the number by the log of the base. For example, to determine how many bits in a computer are required to represent a certain number—say 500—you need to take the log to the base 2, since bits deal in base 2.

```
Bits:=trunc(ln(500)/ln(2))+1;
```

Calendar Functions

A very useful capability for a computer to have is that of dealing with time. The Pascal operating system has some capabilities of dealing with time through the interface to the system clock. However, the clock is more designed to deal with centiseconds, seconds, minutes, and hours, than it is to deal with days, months, and years (although it can do some of this).

This section of the chapter deals with a broader area of timekeeping capabilities, ranging up to time spans of thousands of years.

The Julian Day

The Julian day, named in honor of Julius Caesar, is an astronomical convention representing the number of days that have elapsed since January 1, 4713 B.C. It is nothing more than an arbitrary "zero point" from which dates can be calculated. Since every month/day/year date has a Julian Day number, it becomes quite easy to determine how many days apart events are.

Converting Between Julian Day and Month/Day/Year

The formulae for determining the Julian Day number are these:

$$\text{Day}_{\text{Julian}} = \lfloor 365.25y' \rfloor - \lfloor y'/100 \rfloor + \lfloor y'/400 \rfloor + \lfloor 30.6001m' \rfloor + \text{day} + 1720997$$

where

$$y' = \begin{cases} \text{year} - 1 & \text{if month} \leq 2 \\ \text{year} & \text{if month} > 2 \end{cases}$$

$$m' = \begin{cases} \text{month} + 13 & \text{if month} \leq 2 \\ \text{month} + 1 & \text{if month} > 2 \end{cases}$$

This algorithm is valid only for dates after October 15, 1582, since a 10-day calendar correction was done at that time. If you include the 10-day correction, this October 15, 1582 limit no longer applies.

If an invalid date is sent to the routine, there will be no indication that the number coming back is wrong; you must check for out-or-range conditions yourself.

```
Yr:=Year-ord(Month<=2);
Mo:=Month+1+12*ord(Month<=2);
Julian:=trunc(365.25*Yr)-trunc(Yr/100)+trunc(Yr/400)+trunc(30.6001*Mo)+
 Day+1720997;
```

After converting a month/day/year into a Julian Day number and doing some desired operation, you'll need to convert a Julian Day number back into a month/day/year. These are the formulae you'll need:

$$d' = \text{Day}_{\text{Julian}} - 1720997$$

$$y' = \left\lfloor \frac{d' - 121.5}{365.2425} \right\rfloor$$

$$m' = \left\lfloor \frac{d' - \lfloor 365.25y' \rfloor + \lfloor y'/100 \rfloor - \lfloor y'/400 \rfloor}{30.6001} \right\rfloor$$

$$\text{day} = d' - \lfloor 365.25y' \rfloor + \lfloor y'/100 \rfloor - \lfloor y'/400 \rfloor - \lfloor 30.6001m' \rfloor$$

$$\text{month} = \begin{cases} m' - 13 & \text{if } m' \geq 14 \\ m' - 1 & \text{if } m' < 14 \end{cases}$$

$$\text{year} = \begin{cases} y' & \text{if month} > 2 \\ y' + 1 & \text{if month} \leq 2 \end{cases}$$

In Pascal:

```
D:=Julian-1720997;
Y:=trunc((D-121.5)/365.2425);
Temp:=D-trunc(365.25*Y)+trunc(Y/100)-trunc(Y/400);
M:=trunc(Temp/30.6001);
Day:=Temp-trunc(30.6001*M);
Temp:=M-1-12*ord(M>=14);
Month:=Temp;
Year:=Y+ord(Temp<=2);
```

These two functions allow you to do many desirable things (assume you have a function **Julian** which calculates the Julian day number, and a function **mmddyyyy** which calculates month/day/year):

- *How many days apart were Event A and Event B?*

  DaysApart:=Julian(⟨*Event B date*⟩**)-Julian(**⟨*Event A date*⟩**);**

- *What day of the year is June 18, 1985?*

  DayOfYear:=Julian(⟨*June 18, 1985*⟩**)-Julian(**⟨*January 1, 1985*⟩**)+1;**

- *What will be the date 200 days from today?*

  Date:=Mmddyyyy(Julian(⟨*today*⟩**)+200);**

Day of the Week

The Julian Day number also lends itself nicely to finding out which day of the week on which a particular date fell.

```
DayOfWeek:=(Julian(Month, Day, Year)+1) mod 7+1;
```

This algorithm returns a number in the range 1–7, meaning Sunday–Saturday, respectively.

Leap Year

As mentioned, a leap year is a year in which an extra day is placed at the end of February. The current algorithm, instituted along with the Gregorian Calendar, is this: A year is not a leap year unless it is a multiple of 4, in which case it is, unless it is a multiple of 100, in which case it is not, unless it is a multiple of 400, in which case it is[6].

In Pascal, this is:

```
if Year mod 4<>0 then
  LeapYear:=false
else
  if Year mod 100<>0 then
    LeapYear:=true
  else
    if Year mod 400<>0 then
      LeapYear:=false
    else
      LeapYear:=true;
```

[6] Admittedly complex, but comes very close to the right answer.

Number Base Conversion

Utility functions are available with the Pascal language to simplify some of the conversions between number bases. The three functions **binary**, **octal**, and **hex** convert strings representing numbers in base 2, 8, and 16, respectively, to integers. There are no standard Pascal functions which convert integers to these bases.

For those applications where you must deal with number bases other than 2, 8, or 16, you must create your own conversion routines.

To refresh your memory on conversion of a number in another base to base 10, consider the following. You want to convert 1432 in base 5 to base 10. This is $1\times5^3+4\times5^2+3\times5^1+2\times5^0$, or 125+100+15+2, or 242.

To convert 242 back to base 5, you take successive powers of 5 until the first time a power of five is greater than the original number, then back off one, and this is where you start. For example:

$5^0=1$	1<242, so increment the exponent.
$5^1=5$	5<242, so increment the exponent.
$5^2=25$	25<242, so increment the exponent.
$5^3=125$	125<242, so increment the exponent.
$5^4=625$	625>242, so decrement the exponent; we've found where to start.

Thus, the power of 3 is where we are to start subtracting:

How many 5^3s can be taken from 242?	One; write **1**, do the subtraction.
How many 5^2s can be taken from 117?	Four; write **4**, do the subtraction.
How many 5^1s can be taken from 17?	Three; write **3**, do the subtraction.
How many 5^0s can be taken from 2?	Two; write **2**, do the subtraction.

At this point, the iterations stop, because the original number has been reduced to zero. We've successfully converted 242 in base 10 to base 5; we've written **1432**, which is the original number.

The following code segments illustrate what must be done to convert between base 10 and base *n*. Note that the numbers which are in base 10 are regular integers, and the numbers in other bases are represented as strings, because any base greater than 10 requires letters for the other digits.

Here is an algorithm for converting a positive integer (N) into a string (Strng) representing a number in base n (Base).

```
const
  Chars=                '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'; {base 36 max}
type
  Str32=                string[32];
var
  Power:                integer;
  StrIndex,CharsIndex:  integer;
  Strng:                Str32;
      .
      .
      .
Power:=1;                                 { \   Find out what     }
repeat                                    {  \  number to start   }
  Power:=Power*Base;                      {   > dividing the      }
until Power>N;                            {  /  input parameter   }
Power:=Power div Base;                    { /   by.               }
Strng:=strrpt(' ',32);                    {initialize the result string}
StrIndex:=0;                              {where are we in the string?}
repeat
  CharsIndex:=N div Power;                {get magnitude of "digit" in base n}
  StrIndex:=StrIndex+1;                   {increment character pointer}
  Strng[StrIndex]:=Chars[CharsIndex+1];   {place "digit" in appropriate position}
  N:=N mod Power;                         {subtract digit*Power from N}
  Power:=Power div Base;                  {decrement exponent}
until Power=0;                            {until number goes to 0}
```

Here is an algorithm for converting a string representing an number in base n to an integer:

```
const
  Zero=                 ord('0');
var
  I, Pos, Temp:         integer;
  StrChar:              string[1];
  BadChar:              boolean;
        .
        .
        .
if (Base<2) or (Base>36) then
  begin
  writeln('Error: Base=',Base:0);
  halt(-1);
  end;
BadChar:=false;
for I:=1 to strlen(Strng) do
  begin
  StrChar:=str(Strng,I,1);
  Pos:=strpos(Chars,StrChar);
  if (Pos<1) or (Pos>Base) then
    BadChar:=true;
  end;
if BadChar then
  ⟨error message⟩
else
  begin
  Temp:=0;
  for I:=1 to strlen(Strng) do
    begin
    StrChar:=str(Strng,I,1);
    Pos:=strpos(Chars,StrChar);
    Temp:=Temp*Base+Pos-1;
    end;
  ⟨function name⟩:=Temp;
  end;
```

Random Numbers

In many mathematical and statistical fields of study, there is a need to simulate random events. A random event is an event which does not produce the same outcome every time it occurs under identical circumstances. And, since many events and processes can be mathematically modeled, a computer should be able to model random events.

Technically, a computer is hard-pressed to generate real random sequences, because the one of the requirements of a sequence of random numbers is that the value of any particular number is completely unrelated to its previous and succeeding neighbor. Most computerized random number generators generate random sequences *algorithmically*; that is, the value of each number is somehow derived from the previous one (or n numbers).

Since the definition of "random number sequence" requires that neighboring numbers be unrelated, algorithmic random number generators do not *really* generate random sequences. On the other hand, the sequence of numbers generated by a good algorithmic random number generator passes batteries of randomness tests, therefore the sequences can be considered random. To remove the apparent paradox here—random or not random—computer scientists have called the number sequences generated by algorithmic random number generators "pseudo-random".

Workstation Support of Pseudo-Random Numbers

Your computer has two routines which deal with random sequences. Both of them are exported from module **rnd** (in **SYSVOL:LIBRARY**):

random This procedure takes a random number "seed" and returns the next value in the pseudo-random sequence. The seed of a pseudo-random number generator is a number from which the next value in a sequence of pseudo-random numbers is generated. Typically, this routine is used when the random sequence need not be range-limited. It generates values in the range $0..2^{31}-1$. Its declaration is:

```
procedure random(var seed: integer);
```

The random-number seed "**Seed**" must be initialized prior to use. A good initial value for **Seed** is one with several digits, where the least significant digit is a 1, 3, or a 7.

rand This function takes a pseudo-random number seed and returns a pseudo-random integer in a user-definable range, as well as updating the seed for the next iteration. Its declaration is:

```
function rand(var seed: integer; range: shortint): shortint;
```

The type **shortint** indicates a signed, two's complement, 16-bit integer. It is exported from the **sysglobals** module (in **CONFIG:INTERFACE**):

```
type
  shortint=   -32768..32767;
```

The parameter called **range** allows you to specify the integer range within which the returned pseudo-random integer will be. That is, if you invoke this function with range equal to n, the returned integer will be in the range 0 through $n-1$, inclusive. Obviously, you can add 1 to the function result if you wish the range to be 1 through n. Initial seeds that are good for **random** are also good for **rand**.

Note that the parameter **Seed** will be changed by a call to either **random** or **rand**.

Using the Pseudo-Random Number Generator

Generating a pseudo-random sequence between 0 and $n-1$ or between 1 and n is trivial, if repetition—having the same number more than once—is permitted. It is as easy as:

```
I:=rand(Seed,⟨n⟩);      {Range: 0..⟨n−1⟩, inclusive}
I:=rand(Seed,⟨n⟩)+1;    {Range: 1..⟨n⟩, inclusive}
```

If an arbitrary set of limits is desired, say, you want integers in a pseudo-random sequence between m and n ($m \leq n$), this is as easy as:

```
I:=rand(Seed,⟨n⟩-⟨m⟩+1)+⟨m⟩;
     {Range: ⟨m⟩..⟨n⟩, inclusive}
```

The following program does just that: it generates 100 pseudo-random integers in a user-defined range:

```
program Randoms(input,output);
import rnd;                                  {get the random number routines}
var
  Seed, Rmin, Rmax, I:  integer;
begin
Seed:=12345;                                 {initialize the random number seed}
write('Range for random numbers: ');         {ask for...}
readln(Rmin,Rmax);                           {...and receive the range limits.}
for I:=1 to 100 do                           {100 times...}
  write(rand(Seed,Rmax-Rmin+1)+Rmin:0,' ');      {...write a number between...}
                                                 {...Rmin and Rmax.}
end.
```

If repetition is not allowed, it is not quite as straightforward, although it is not difficult. Note that given the seed of 1234, **random** will generate all integers $0..2^{31}-1$ before repeating.

An example of generating a random sequence without repetition is shuffling a deck of cards. No matter how poorly or how well the randomness is applied, there will never be more than one ace of spades[7], or seven of clubs, etc.

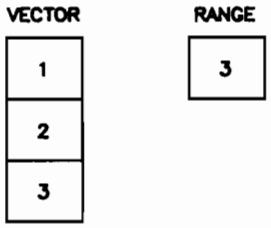
The following example concerns generating a pseudo-random number sequence of arbitrary size *without* repetition. The routine in this example generates n pseudo-random integers between 1 and n, although it could easily be modified to generate fewer than n integers in the range 1 to n, or to generate integers between m and n.

[7] We are dealing with a regular deck of cards here, not a pinochle deck. Our deck has Ace through King of each suit.

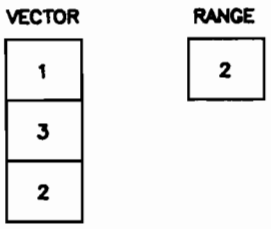
A Shuffling Algorithm

Generating a pseudo-random list of non-repeating numbers is not difficult. Let's go through the algorithm by hand to generate a list of 3 pseudo-random numbers. There needs to be a vector—a one-dimensional array—3 elements long, through which the shuffled integers will be returned to the calling routine. In addition to this, we need a temporary storage area of type **integer**. There also needs to be a variable called **Range** which specifies the maximum value the random number can be when selecting elements from the array.

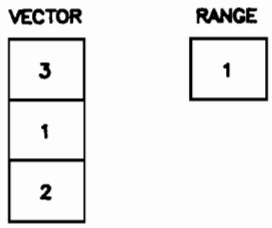
1. Define `Vector[I]:=I`. The setup looks like this:



2. Pick a pseudo-random number, ⟨*rnd*⟩, in the range 1 through **Range**, which is currently 3. Let's say ⟨*rnd*⟩ is 2.
3. Switch the values of `Vector[Range]` and `Vector[`⟨*rnd*⟩`]`. We have now defined the pseudo-random number in the element of **Vector** specified by **Range** (now 3). Decrement **Range**. The setup now looks like this:



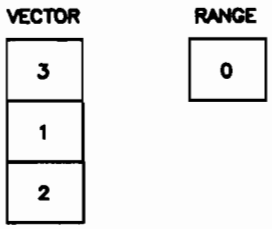
4. Pick a pseudo-random number in the range 1 through **Range**, which is currently 2. Let's say ⟨*rnd*⟩ is 1.
5. Switch the values of `Vector[Range]` and `Vector[`⟨*rnd*⟩`]`. We have now defined the pseudo-random number in the element of **Vector** specified by **Range** (now 2). Decrement **Range**. The setup now looks like this:



6. The final step of this algorithm is virtually a no-op. It is driven by "Pick an integer between 1 and 1, inclusive." However, for the sake of completeness, we will go through it.

   Pick a pseudo-random integer in the range 1 through **Range**, which is currently 1. Obviously, ⟨*rnd*⟩ must have the value of 1.

7. The switching of the values in **Vector[Range]** and **Vector[**⟨*rnd*⟩**]** doesn't change anything this time. Decrement **Range**. The setup looks like this:



8. **Range** has been reduced to 0, so we are done. Return the array to the calling routine.

The Shuffling Routine

Putting the above algorithm into Pascal is quite simple, as the following example shows:

```
program Shuffle_(output);
import rnd;                              {get RANDOM and RAND}
var
  Vector1:      array [1..10] of integer;
  Vector2:      array [1..20] of integer;
  I:            integer;
$page$  {*************************************************************}
procedure Shuffle(var Vector: array [Lo..Hi: integer] of integer);
var
  Temp:         integer;                 {temporary storage area}
  Seed:         integer;                 {pseudo-random number seed}
  Range:        integer;                 {maximum random number}
  I, J:         integer;
begin
Seed:=1234567;                           {initialize the random number seed}
for I:=Lo to Hi do                       {initialize the temporary array}
  Vector[I]:=I;
Range:=Hi;                               {pick from whole thing the first time}
for I:=Lo to Hi do
  begin
  J:=rand(Seed,Range)+1;                 {where does next element go?}
  Temp:=Vector[Range];                   { \  Switch locations  }
  Vector[Range]:=Vector[J];              {  > of Vector[Range]  }
  Vector[J]:=Temp;                       { /  and Vector[J].    }
  Range:=Range-1;                        {reduce the choice range}
  end;
end;
$page$  {*************************************************************}
begin
Shuffle(Vector1);
writeln('Shuffled vector: ');
write('  ');
for I:=1 to 10 do write(Vector1[I]:0,strrpt(', ',ord(I<10)));
writeln;
writeln;
Shuffle(Vector2);
writeln('Shuffled vector: ');
write('  ');
for I:=1 to 20 do write(Vector2[I]:0,strrpt(', ',ord(I<20)));
writeln;
end.
```

There are several features of note in the above example:

- The array to be filled with non-repetitive pseudo-random numbers is passed to the shuffling routine as a conformant array. The routine automatically adjusts its behavior to deal with whatever size array was passed to it. Note that since the array is passed as a conformant array, it may not be portable to other Pascal systems. (See the chapter "Data Structures" for more information on conformant arrays.)
- Using the step-function capability provided by using `ord(`⟨*boolean value*⟩`)` as a number. (See the section "Step Functions" in this chapter for more information on these.)

The following are references which contain further information on numeric computation.

Coonen, Jerome T.; "An Implementation Guide to the Proposed Floating Point Standard", *Computer Magazine*, Jan. 1980.

Cody, William J. Jr. and William Waite; *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

Sterbenz, Pat H.; *Floating Point Computation*, Prentice Hall, 1974.

Signum Newsletter, Oct 1979.

String Manipulation 14

Introduction

It is often desirable to store non-numerical information in the computer. A word, a name or a message can be stored in the computer as a **string**. Any sequence of characters, both displayable and non-displayable, may be used in a string. Apostrophes ('), or single quote marks, are used to delimit the beginning and end of a string literal (see examples below). The following are valid string variable assignments.

```
A:='COMPUTER';
Fail:='The test has failed.'#7;  {the "#7" is a CTRL-G (bell) }
File_name:='INVENTORY';
TEST:=str(Fail,5,4);
```

The variable (the left-hand side of the assignment) gets the string value specified by the right-hand side of the assignment.

The *length* of a string is the number of characters in the string. In the previous example, the length of `A` is 8 since there are eight characters in the literal `'COMPUTER'`; you don't count the quotes, since they are only used to delimit the beginning and end of the literal.

Pascal (as implemented in the Pascal Language System) allows the dimensioned length of a string to range from 1 to 255 characters. The current length (number of characters in the string) may range from zero to the dimensioned length. A string of zero characters is called a "null string" or an "empty string". An error results whenever you try to assign a string variable more characters than its dimensioned length.

Special Cases of String Assignment

A string may contain any character. There are three special cases when trying to assign a literal to a string.

- The quote mark itself,
- Control characters (`ord`<32), and
- The upper half of the character set (`ord`>=128).

Getting a Quote Into the String

To get the quote mark (or "apostrophe") itself into the string requires two quotes in succession (or the "pound-sign" notation described below):

```
Quoted:='The time is ''NOW''.';
Apostrophe:='''';
writeln(Quoted);
writeln(Apostrophe);
```

Produces:

```
The time is 'NOW'.
'
```

Getting a Control Character Into a String

To get control characters whose ordinal value is less than 32 into a string, you put a character or an integer[1] after a pound sign (a "#"). Say that you wanted a string to contain an "A", a carriage return, and a "B". You could type:

```
Strng:='A'#M'B';
```

The pound sign and the character following are converted into `chr(ord(`⟨*character*⟩`) mod 32)`. An ASCII table will provide information on what values to use.

Note that these characters cannot be *inside* quote marks, or you will end up with just those characters. For example, if the two inner quote marks in the above example were removed, the string would consist of an "A", a "#", an "M", and a "B".

In the same way as a non-numeric character can follow a pound sign, a number can, too. To get the same string as the above example, you could type:

```
Strng:='A'#13'B';
```

Again, notice that the pound sign and its number must be outside of quotes.

Getting "Other" Characters Into a String

The "pound-sign-character" method mentioned above is limited to creating characters whose `ord` is less than 32. The "pound-sign-integer" method has no such restriction; it can create any character between `chr(0)` and `chr(255)`, inclusive.

For example, if your display supports underlining text, you can cause a string to contain its own underline on/off characters:

```
Strng:=#132'This is underlined.'#128;
```

[1] If you put an integer after the #, you are not limited to characters whose **ord** is less than 32. See the next section.

Declaring String Variables

The following statements may be used to declare a string:

```
type
  Str20=              string[20];
var
  MyString:           Str20;
```

or

```
var
  MyString:           string[20];
```

String Length

A string may be declared (dimensioned) to any length between 1 and 255 characters, inclusive. The `var` statement declares and reserves storage for string variables.

```
const
  ShortStringLength= 4;
type
  ShortStringType=    string[ShortStringLength];
var
  ShortString:        ShortStringType;
  LongString:         string[255];
```

Strings that have been allocated but not assigned can contain anything; there is no automatic housekeeping done. Therefore, string variables should be initialized to some known state before use (e.g. `longstring:=''`).

String Storage in Memory

Strings, as all other Pascal variables, must have space reserved before assignment. That space reserved consists of one length byte, followed by as many characters as specified in the declaration (the length byte is a one-byte area at the beginning of every string which indicates, in its eight bits, the current length of the string). The storage area is aligned along an even-byte boundary. Thus, a variable declared as `string[6]` will consume 8 bytes: the six bytes desired, the length byte, and another byte for padding to an even-byte boundary.

String Arrays

Strings, like any other data type in Pascal, can be incorporated in arrays and records. Large amounts of text are easily handled in arrays. For example:

```
var
  BigArray:           array[1..100] of string[80];
```

This statement reserves storage for 100 lines of 80 characters per line. Each string in the array can be accessed by an index. For example:

```
writeln(BigArray[27]);
```

Prints the 27th element (string) in the array.

Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string (plus one, for the current length, plus possibly another one for the even-byte-boundary pad character), string arrays can quickly use a lot of memory.

Evaluating Expressions Containing Strings

Evaluation Hierarchy

Evaluation of string expressions is simpler than evaluation of numerical expressions. The two allowed operations are concatenation and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses (functions, which require parenthesized parameters, are included here).
Low	Concatenation

String Concatenation

Two separate strings are joined together by using the concatenation operator "+". The following program segment combines two strings into one.

```
One:='WRIST';
Two:='WATCH';
Concat:=One+Two;
writeln(One,' ',Two,' ',Concat);
```

Prints:

```
WRIST WATCH WRISTWATCH
```

The concatenation operation, in the third line, appends the second string to the end of the first string. The result is assigned to a third string. An error results if the concatenation operation produces a string value that is longer than the dimensioned length of the string variable to which it is being assigned.

To increase the readability of certain programs, parentheses can be used to force concatenation in a particular order. Note that the outcome result will be the same with or without parentheses, since all string operators (there is only the one) are associative. This is different from numeric expression evaluation, where there are several different operations, having different associativity and distributive properties.

```
CombinedString:=Strng1+(Strng2+Strng3);
```

Relational Operations

The relational operators used for numeric expression evaluation can also be used for the evaluation of strings and string literals. Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined.

The following examples show some of the possible tests.

```
'ABC' = 'ABC'                    True
'ABC' = ' ABC'                   False

'ABC' < 'AbC'                    True
'6'   > '7'                      False
'60'  > '7'                      False

'long' <= 'longer'               True
'RE-SAVE' >= 'RESAVE'            False
```

Any of these relational operators may be used: <, >, <=, >=, =, <>.

The outcome of a relational test is based first on the characters in the strings and, second, on the length of the strings. For example:

```
'BRONTOSAURUS' < 'CAT'
```

This relationship is true since the letter "C" is higher in ASCII (or *ordinal*) value than the letter "B". However, in the following example, the string length *is* taken into account:

```
'HIPPO' < 'HIPPOPOTAMUS'
```

In this case, all the characters match up through the point at which one string ends. At this point, the shorter string is considered the lesser.

String Functions

Several intrinsic functions are available in HP Pascal for the manipulation of strings. These functions include:

- Extracting substrings
- Determining string length and maximum string length
- Locating substrings within strings
- Conversion between string and numeric values
- Conversion between strings and packed arrays of characters
- Trimming off leading and/or trailing blanks
- Repeating strings zero or more times

Substrings

Using the string function **str**, you can extract a portion of a string, called a *substring*, from the source string. A substring may comprise all or just part of the original string. The **str** function requires three parameters:

- The source string expression
- The starting index of the substring
- The substring length

For example, assuming **Strng** is a string variable dimensioned to a maximum length of 20, and that it currently has the 16-character value of **'abcdefghijklmnop'**:

str(Strng,3,4)	specifies a substring of **Strng** starting at the third character and extending for 4 characters: **'cdef'**.
str(Strng,16,1)	specifies a substring of **Strng** starting at the sixteenth character and extending for 1 character: **'p'**.
str(Strng,3,0)	specifies a substring of **Strng** starting at the third character and extending for zero characters: **''**.
str(Strng,39,4)	specifies a substring of **Strng** starting at the thirty-ninth character and extending for 4 characters: **Error!**
str(Strng,60,0)	specifies a substring of **Strng** starting at the sixtieth character and extending for zero characters: *No* error.

Except for null substrings, the integer expression specifying the starting position of the substring must be in the range 1 to the current length of the string.

The **str** function may appear only on the right side of an assignment statement.

Current Length of a String

The "length" of a string is the number of characters in the string. The **strlen** function returns an integer whose value is equal to the current string length. The range is from 0 (given by the null string) through the dimensioned length of the string. For example:

```
write(strlen('HELP ME'));
Strng:='Greetings!';
writeln(strlen(Strng));
```

Prints: 7 10

Maximum Length of a String

This function returns the maximum length a string can legally be. This is its length as specified in its declaration, e.g., **string[80]**.

The **strmax** function can be used to avoid run-time errors which would occur from string overflows. For example:

```
if strlen(Strng)+strlen(Addendum)>strmax(Strng) then
  writeln('String would overflow.  Append operation not performed.')
else
  Strng:=Strng+Addendum;
```

Substring Position

The "position" of a substring within a string or string literal is determined by the **strpos** function. The function returns the value of the starting position of the *first occurrence of the substring* or zero if the entire substring was not found. For instance:

```
writeln(strpos('APPEAR','DISAPPEARANCE'));
```

prints 4, because the substring **'APPEAR'** is found in the string literal **'DISAPPEARANCE'**, and it starts in fourth character position.

The compiler option **$switch_strpos$** reverses the interpretation of the arguments in a **strpos** call. This brings the order of arguments into agreement with the HP Pascal Standard (which is also in agreement with the HP BASIC definition of **POS**, a similar function). That is, if **$switch_strpos$** is in effect, the above example would have been coded:

```
writeln(strpos('DISAPPEARANCE','APPEAR'));
```

If **strpos** returns a non-zero value, the entire substring occurs in the first string and the value specifies the starting position of the substring.

The **$switch_strpos$** directive, if it is used, must appear at the beginning of the program. It sets (it doesn't complement) an internal flag which specifies that the interpretation order of **strpos** parameters should conform to the HP standard; thus, multiple occurrences of **$switch_strpos$** do not keep toggling the interpretation order.

Sometimes, you may not care *where* a substring is in a string, you need to find out only *if* it is in the string. Again, the **strpos** function is useful:

```
$switch_strpos$
      .
      .
      .
var
  MasterList:          string[255];
  Item:                string[10];
  Found:               boolean;
      .
      .
      .
Found:=(strpos(MasterList,Item)>0);
if Found then ...
```

Note that **strpos** returns only the *first* occurrence of a substring within a string. By extracting a substring, and indexing through it, the **strpos** function can be used to find any occurrence or all occurrences of a substring. The following algorithm uses this technique to find any specified substring from a source string.

Assume that the source string—that string to be searched—is called **Source**, and that the substring you are looking for is called **Pattern**. Further, assume that the occurrence of the substring you are looking for is an integer called **Occurrence**. In other words, if you are looking for the third occurrence of "**is**" in the string "**This is the Mississippi**", you would set **Source** to "**This is the Mississippi**", **Pattern** to "**is**", and **Occurrence** to 3.

Note that in this algorithm, we are not permitting overlapping occurrences of the pattern sought. Thus, there is only one occurrence of "issi" in "Mississippi"; it starts in character 2. The occurrence starting at character 5 is not considered because the search resumes at character 6.

The following steps are required:

1. Find, in the whole of **Source**, the position of the first occurrence of **Pattern**. Place this value in the integer **Pos**.
2. If **Pattern** exists in the section of **Source** scanned (and if we haven't found the one we're looking for yet), do the following:
    a. Make note of the fact that you've found an(other) occurrence of **Pattern**.
    b. If we've found the one we're looking for, return the location of **Pattern** within the section of **Source** we just searched. If we haven't found the one we're looking for, search **Source** *from the first point* another occurrence of **Pattern** could exist, and, if it exists, note its position in **Pos**. What is meant by "from the first point another occurrence could exist" is this: the second occurrence of a string cannot occur (by our rules) until *after* the first occurrence ends. Thus, skip over the part of the string occupied by the characters before **Pos**, as well as the entire length of the first occurrence of the pattern.
    c. Go to Step 2.

3. We got out of the Step 2 loop because either (1) no more occurrences were found, or (2) we found the occurrence we were looking for. If we found the one we're looking for, return the location of `Pattern` within the whole of `Source`. If we didn't find the one we're looking for, return zero; the specified occurrence does not exist.

In Pascal, the following code segment accomplishes the desired task. Assume that this is part of a string function whose declaration looks like this:

```
function strpos2(Source, Pattern: Str255; Occurrence: integer): Str255;
```

where `Str255` is a type specifying `string[255]`.

```
if Occurrence=1 then                    {if looking for the first one...}
  strpos2:=strpos(Source,Pattern)           {...use the system routine.}
else
  begin                                 {otherwise...}
  Start:=1;                             {where to start search in Source}
  Found:=0;                             {how many have we found?}
  Plength:=strlen(Pattern);             {length of pattern searched for}
  Done:=false;                          {done yet?
  Pos:=strpos(Source,Pattern);          {search for Pattern in Source}
  while (Pos>0) and not Done do         {if we're still going...
    begin
    Found:=Found+1;                     {eureka! another one!}
    if Found=Occurrence then            {the one we're looking for?}
      Done:=true                        {yes; quit}
    else
      begin                             {no...}
      Start:=Start+Pos+Plength-1;       {update search starting position}
      Pos:=strpos(str(Source,Start,strlen(Source)-Start+1),Pattern);
                                        {where in THIS PART is Pattern?}
      end;  {else}
    end;  {while}
  if Found<Occurrence then                  {did we exit loop for failure?}
    strpos2:=0                          {yep...}
  else
    strpos2:=Start+Pos-1;               {no, we found the one sought}
  end;  {Occurrence>1}
```

As each occurrence is found, the new value of `Start` specifies the remaining portion of the string to be searched.

String-to-Numeric Conversions

The **strread** function reads data from a string much as **read** reads data from a file. **strread** can do anything with a string that a **read** can do with a file with the exception of end-of-line-related operations. When reading an integer or an enumerated-type item, the string must evaluate to a valid value or error −10 will result[2].

```
error -10: bad input format
```

Note that enumerated types include the **boolean** type, defined **boolean=(false,true)**.

The **strread** procedure requires at least four parameters. They are:

strread(⟨*string to read from*⟩,⟨*starting position*⟩,⟨*next position to read*⟩,
⟨*variable 1*⟩, ... ,⟨*variable n*⟩);

A description of these parameters follows:

⟨*string to read from*⟩ The string expression from which certain characters are to be read into a number (either integer or real values can be read), an enumerated type, etc.

⟨*starting position*⟩ The starting index. This integer expression specifies where in the source string read should begin. It must be in the range from 1 to the current length of the source string.

⟨*next position to read*⟩ The "next" character. Upon completion of the **strread** procedure, this integer variable contains the position in the string of the next character to be read after all the variables (see variable below) have been assigned.

For example, if you were reading from the string **'123 456'** into a single variable, and the starting index was 1, this integer, specifying the "next" character would become 4. This is because after reading the characters **'123'** and converting them to the integer 123, character 4 is the next one to process. So, the next time through the loop, the *second* parameter should be set to 4, reading started there, and "next" would be assigned 8. Observe:

```
program StringRead(output);
var
  Strng:                string[80];
  Start, Next:          integer;
  Number:               integer;
  Color:                (Red, Green, Blue);
  Truth:                boolean;
begin
Strng:='123 red true 45 green true 6789 blue false ';
Start:=1;
while Start<>strlen(Strng) do
  begin
  strread(Strng, Start, Next, Number, Color, Truth);
  writeln(Number, ' ', Color, ' ', Truth);
  Start:=Next;
  end;
end.
```

[2] The system reports **escapecode** = −10, but technically, this only means that some kind of I/O error took place, and thus **ioresult** is nonzero. At this point, **ioresult** is examined by the system (it equals 14) in order for the system to print the "bad input format" message.

This program prints:

```
  123 RED TRUE
   45 GREEN TRUE
 6789 BLUE FALSE
```

⟨*variable*⟩ After the first three parameters specified above, there must exist one or more variable names, into which the **strread** procedure places values read from the string.

A number returned by the **strread** function will be converted from scientific notation when necessary. For example, where **NumValRead** is real:

```
strread('123.4E3',1,NextChar,NumValueRead);
writeln(NumValueRead);
```

Prints: 1.23400E+005

The following program converts a fraction into its equivalent decimal value. It has no checks on input format, and may give error messages if the input is bad.

```
$switch_strpos$
program ValFrac(input, output);
var
  Fraction:                              string[255];
  Delimiter, Numerator, Denominator:     integer;
  NextChar:                              integer;
begin
write('Enter a fraction (e.g., 3/4): ');
readln(Fraction);
Delimiter:=strpos(Fraction,'/');
Fraction[Delimiter]:=' ';       {remove slash so STRREAD will work}
strread(Fraction,1,NextChar,Numerator, Denominator);
Fraction[Delimiter]:='/';       {put it back so the fraction looks right}
writeln(Fraction,' = ',Numerator/Denominator:30:15);
end.
```

Similar techniques can be used for converting feet and inches to decimal feet, or hours and minutes to decimal hours.

Character-to-Numeric Conversions

The **ord** function converts a single character into its equivalent numeric value; that is, its ASCII value. The number returned is in the range 0 to 255. For example:

```
writeln(ord('A'));
```

Prints: 65

This use of **ord** is implementation dependent.

The next program prints the value of each character in a name.

```
program Ord_(input, output);
var
  Strng:        string[255];
  I:            integer;
begin
write('Text: ');
readln(Strng);
for I:=1 to strlen(Strng) do
  write(ord(Strng[I]):0,' ');
writeln;
end.
```

Entering the name "JOHN" will produce the following.

```
74 79 72 78
```

Numeric-to-String Conversions

The **strwrite** function converts the value of a numeric or enumerated-type expression into a string of characters (again, "enumerated type" includes **boolean**). A string representing a number contains the same numeric characters (digits, decimal point, and/or exponent) that appear when the numeric variable is printed. For example:

```
strwrite(Strng,1,I,1000000,' ',true);
writeln(1000000,' ',true,' ',Strng)
```

Prints: `1000000 TRUE     1000000 TRUE`

A function could be defined which takes a real number as an argument (that way, integers could be passed to it, too) and returns an appropriate-looking string. You probably would **strwrite** the number into the string with a large enough format specifier so as to avoid scientific notation until absolutely necessary. For example:

```
strwrite(Strng,1,Next,RealNumber:31:15);
```

After the number (which can have up to 15 digits on each side of the decimal point before resorting to scientific notation) is in the string, you could then remove the leading spaces, trailing zeroes, and possibly a trailing decimal point from the string, and you're done.

Note that performing a **strwrite** on an uninitialized string may give unpredictable results. Before doing a **strwrite**, be sure that the string is empty, or contains useful data.

Numeric-to-Character Conversions

The chr function converts a number into an ASCII character. The number must be an integer, and the value must be in the range 0 through 255. For example:

```
writeln(chr(97),chr(98),chr(99));
```

Prints: abc

The next program converts the numeric values in an array constant to characters.

```
program Chars(input,output);
type
  CharArrayType=array[1..15] of 0..255;
const
  CharArray=    CharArrayType
                  [34,130,89,111,117,32,103,111,116,32,105,116,33,128,34];
var
  I:            integer;
begin
for I:=1 to 15 do
  write(chr(CharArray[I]));
writeln;
end.
```

String Repeat

The strrpt function returns a string created by repeating the specified string a given number of times.

```
writeln(strrpt('* *',10));
```

Prints: * ** ** ** ** ** ** ** ** ** *

This function can be used when centering titles. The algorithm is:

1. Subtract the length of the title from the width of the printer/display device to find out how much space is *not* taken up by the title.
2. Divide this amount by two to find the amount of space which should be on the left side of the title.
3. Print that amount of space, followed by the title. The title will be centered.

For example:

```
Title:='⟨any text, as long as it's narrower than the printer⟩';
writeln(strrpt(' ',(PrinterWidth-strlen(Title)) div 2),Title);
```

Note that this will work in the intuitive way for all titles shorter than the printer is wide, *as long as there are no unprintable characters in it* (for example, underlining the title requires a chr(132) at the beginning and a chr(128) at the end). To take care of this case, just subtract 1 from the length of the title for every character not in the range ' '..'■', or chr(32)..chr(127).

Trimming a String

The `strltrim` and `strrtrim` functions return a string with all left (leading) and right (trailing) blanks (ASCII spaces) removed, respectively.

```
writeln('*',strltrim('    1.23    '),'*');
writeln('*',strrtrim('    1.23    '),'*');
writeln('*',strltrim(strrtrim('    1.23    ')),'*');
writeln('*',strrtrim(strltrim('    1.23    ')),'*');
```

Prints:

```
*1.23    *
*    1.23*
*1.23*
*1.23*
```

Combining Strings

There are several ways to combine multiple strings into a single string:

Concatenation	This operator works with any number of string expressions.
`strappend`	This procedure appends one string expression to a string.
`strinsert`	This procedure inserts one string into another at any point.

Concatenation

Note that the concatenation operator is just that—an *operator*—which means that it is placed between the operands it is to combine (infix order). As mentioned, it can combine any number of string expressions:

```
Concatenation:='String 1'+'String 2';
All:=A+'another'+(strrpt(' ',Width div 2)+str(Strng,2,5))+'and '#7+Strng;
```

Appending Strings

This procedure requires a string variable as the first parameter; the second parameter may be any kind of a string expression. Upon completion, the string variable has the value it had before with the string expression concatenated to it at the right end.

```
String:='Pascal ';
strappend(Strng,'strings');
writeln(Strng);
```

Prints: `Pascal strings`.

Inserting in the Middle

This procedure requires a string variable, a string expression, and an index into the string variable. The procedure causes the string expression to be inserted into the string variable at the specified (index) point.

```
Strng:='Thus';
strinsert(Strng,'esau',3);
strinsert(Strng,'r',7);
writeln(Strng);
```

Prints: `Thesaurus`.

Replacing/Appending and Conversion Between Strings and PACs

There is another string-related procedure called **strmove** which allows several operations to take place:

- You can append characters to the end of a string (e.g., "**bring**"→"**bringing**");
- You can replace characters in a string one-for-one with other characters (for example, "**inside**"→"**in the**");
- Both of the above—replacing characters and extending the string—simultaneously (e.g., "**sheaf**"→"**sheaves**");
- Convert a PAC variable[3] to a string, and vice versa, without having to move the characters one at a time.

The procedure **strmove** takes five parameters. First, the number of characters to move from the source to the destination. Next, for both the source and the destination, the entity and the index into the entity. For example:

```
strmove(Nchars,SourceExpr,SourcePos,DestVar,DestPos);
strmove(4,A,5,B,6);          { Move 4 characters, starting with A[5], into }
                             { B, starting at position 6.                  }

S:='pal';
strmove(1,'that',4,S,2); {Move 1 character, starting with the 4th character}
                         {of 'that', into S ('pal'), starting at position 2}
```

Reducing Strings

You can delete characters in a string in any of several ways:

- Deleting one or more characters from the "middle" (the "middle" could extend to either end, or conceivably to both ends, deleting the whole thing),
- Deleting one or both ends of the string simultaneously (the deleted portions could conceivably touch, deleting the whole thing),
- Trimming leading or trailing blanks (we saw this before).

Deleting Characters from the Middle

The **strdelete** procedure deletes a specified number of characters from the middle of a string. You specify the string to be reduced, where to start deleting characters, and how many characters to delete:

```
Strng:='strings';                  {Strng now equals 'strings'.}
strdelete(Strng,7,1);              {Strng now equals 'string'.}
strdelete(Strng,2,2);              {Strng now equals 'sing'.}
strdelete(Strng,strlen(Strng),1);  {Strng now equals 'sin'.}
strdelete(Strng,1,13 mod 4);       {Strng now equals 'in'.}
strdelete(Strng,2,2);              {Strng now equals ''.}
```

[3] A "PAC" variable is a **packed array [1..*n*] of char**. Note that the array must be packed, and the first subscript of the array must be 1.

Deleting Both Ends

In order to delete zero or more characters from both ends of a string simultaneously, you have to get a trifle cagey. You don't really *delete the ends*, you *retain the middle*; that is, you assign the variable the value of a substring from the middle:

```
Strng:='antidisestablishmentarianism';
Strng:=str(Strng,8,9);                    {Strng now equals 'establish'.}
```

Trimming Blanks

We discussed these functions before (see "Trimming a String", above). The functions `strltrim` and `strrtrim` remove leading and trailing blanks, respectively.

User-Defined String Functions

Although there are several string functions available in Series 200/300 Pascal, there are several more which are not supplied with the language which can be very useful.

Note

When creating special string functions, testing should include passing the null string (`''`) to the function. The null string is a valid string and may get passed to the function.

Case Conversion

Often, you may want to convert the letters in a string—keyboard input, for example—to uppercase or lowercase letters. This is quite an easy thing to do, since the ASCII values (the "`ord`") of uppercase letters differ from the ASCII values of the corresponding lowercase letters by 32. Below is the algorithm for converting to uppercase:

```
for I:=1 to strlen(Strng) do
  begin
  Character:=Strng[I];                  {avoid subscripting multiple times}
  if (Character>='a') and (Character<='z') then
    Strng[I]:=chr(ord(Character)-32);
  end;  {for I}
```

The algorithm for converting to lowercase is very similar; you just *add* 32, rather than subtracting 32:

```
for I:=1 to strlen(Strng) do
  begin
  Character:=Strng[I];                  {avoid subscripting multiple times}
  if (Character>='A') and (Character<='Z') then
    Strng[I]:=chr(ord(Character)+32);
  end;  {for I}
```

Note: both of these algorithms can be sped up by using the compiler option **`$partial_eval$`**.

Also note that both algorithms are implementation dependent (although they should work for all computers which use the ASCII character set).

String Reverse

A string reversal function returns a string created by reversing the sequence of characters in the given string. For example, reversing `'abc'` results in `'cba'`. Again, the algorithm is elementary:

```
Length:=strlen(Strng);
LengthPlus1:=Length+1;        {avoid adding 1 every iteration...}
for I:=1 to Length div 2 do
  begin
  Temp:=Strng[I];
  RightChar:=LengthPlus1-I;
  Strng[I]:=Strng[RightChar];
  Strng[RightChar]:=Temp;
  end;
```

Note that when the string has an even number of characters in it, all appropriate pairs of characters are switched in position, but when the string has an odd number of characters in it, the middle character is never addressed. This is fine; it doesn't *need* to be addressed, because the middle character is the middle character, regardless of which end you start from.

If you incorporated the above algorithm into a function called **strrev**, the following statement:

```
writeln(strrev('Straw? No, too stupid a fad. I put soot on warts.'));
```

would print:

```
.straw no toos tup I .daf a diputs oot ,oN ?wartS
```

A common (but inefficient) use for the string reversal function is to find the last occurrence of an item in a string. Assume again that a function **strrev** is defined which returns the reversed argument.

```
$switch_strpos$
      .
      .
      .
var
  Strng, LastItem:  string[80];
  Delimiter:        string[1];    {must be a string; STRPOS doesn't like CHAR}
  LastDelim:        integer;
      .
      .
      .
Strng:='Now is the time for all good men to come to the aid of their country.';
Delimiter:=' ';
LastDelim:=strlen(Strng)-strpos(strrev(Strng),Delimiter)+1;
LastItem:=str(Strng,LastDelim+1,strlen(Strng)-(LastDelim+1)+1);
writeln('The last item is "',LastItem,'".');
```

Displays: `The last item is "country.".`

Search-and-Replace Operations

A commonly used operation when dealing with strings is this: "I want to replace each one of *these* in this string with one of *those*." This very useful function entails several sub-operations:

1. Find, in the main string, the first occurrence of the "old" string (that string which is to be replaced, hereafter called ⟨*old*⟩).
2. Delete that occurrence of ⟨*old*⟩, and insert one occurrence of the string which is to replace it (hereafter called ⟨*new*⟩). Note that this must be a deletion followed by an insertion; it cannot be a "direct replacement", because ⟨*new*⟩ may be a different length than ⟨*old*⟩.
3. Starting from the first character *after the end* of the newly-inserted ⟨*new*⟩, search for another occurrence of ⟨*old*⟩. You cannot just start searching again from the beginning of the main string, because it is perfectly legal for ⟨*new*⟩ to contain one or more occurrences of ⟨*old*⟩. If this was the case, searching from the beginning of the main string would result in either (1) an infinite loop, if ⟨*new*⟩=⟨*old*⟩, or (2) a string overflow error if `strlen(`⟨*new*⟩`)>strlen(`⟨*old*⟩`)`.
4. Repeat steps 2 through 3 until there are no more occurrences of ⟨*old*⟩ in the searchable section of the main string.

Taking these things into account, the following code segment accomplishes the desired task. Assume that the type `Str255` has been defined as `string[255]`, and that `$switch_strpos$` is in effect. `Strng` is the main string in which the replacements take place; `Old` and `New` have their intuitive meanings.

```
$switch_strpos$
      .
      .
      .
var
  LengthOfStrng:        integer;
  LengthOfOld:          integer;
  LengthOfNew:          integer;
  Pos, Temp:            integer;
      .
      .
      .
if (Strng='') or (Old='') then {do nothing}
else begin
  LengthOfStrng:=strlen(Strng);      { \                              }
  LengthOfOld:=strlen(Old);          {  > Things go faster this way... }
  LengthOfNew:=strlen(New);          { /                              }
  Pos:=strpos(Strng,Old);
  while Pos>0 do begin
    Strng:=str(Strng,1,Pos-1)+New+
     str(Strng,Pos+LengthOfOld,LengthOfStrng-(Pos+LengthOfOld)+1);
    LengthOfStrng:=LengthOfStrng-LengthOfOld+LengthOfNew;
    Temp:=Pos+LengthOfNew;
    Pos:=strpos(str(Strng,Temp,strlen(Strng)-Temp+1),Old);
    if Pos>0 then Pos:=Pos+Temp-1;
  end;  {while}
end;  {else begin}
```

Sections of Strings

This section just discusses how to get the more common parts of strings in the easiest way.

Left Part

To get the left part of a string, up to and including character *n*, do the following:

```
Strng:=str(Strng,1,n);
```

Right Part

To get the right part of a string, from character *n* to the end, do the following:

```
Strng:=str(Strng,n,strlen(Strng)-n+1);
```

Middle Part: Point A through Point B

To get the middle part of a string when you know the starting and ending positions (*start* and *finish*, respectively), do the following:

```
Strng:=str(Strng,start,finish-start+1);
```

Programming With Files 15

Introduction

The File System of your Pascal system organizes and accesses information which is stored in files on mass storage devices. This section describes how the information is organized and accessed. It consists of the following major discussions:

- Overview of mass storage, including descriptions of files and volumes.
- Techniques for using item-oriented files.
- Techniques for using line-oriented files.
- More details of Pascal Workstation files.

If you are already familiar with files and volumes, you may want to just scan the drawings in the overview section. The "techniques" sections give several examples and some good advice about using Workstation files. The "More Details" section is a more in-depth look at the facts about the Workstation File System.

Overview of Files

This section describes several concepts relating to the use of mass storage files.

Primary versus Secondary Storage

Your computer has built into it a substantial amount of very high-speed memory called random-access memory, or RAM. This memory is called *primary storage* to distinguish it from mass storage, which is called *secondary storage*. Normally, data processed by the computer must be first placed in primary storage. (The term "data" is used here broadly to signify any information processed by the computer; thus, programs are data, too.)

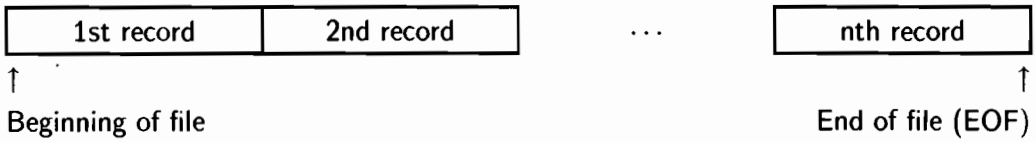
Information not immediately needed by the computer is kept in secondary storage. Mass storage devices are typically less expensive to maintain, are non-volatile (information is not lost when power is removed), and have much greater capacities (hence the term "mass").

What Is a File?

Good question. A file is a logically defined storage area set aside for the temporary or permanent storage of a collection of similar data items. Files consist of two main parts:

- A description (a name, type, size, etc. in a mass storage *directory*).
- Some data (the actual information it contains).

Here is a conceptual drawing of the data part of a file, showing its sequential nature.

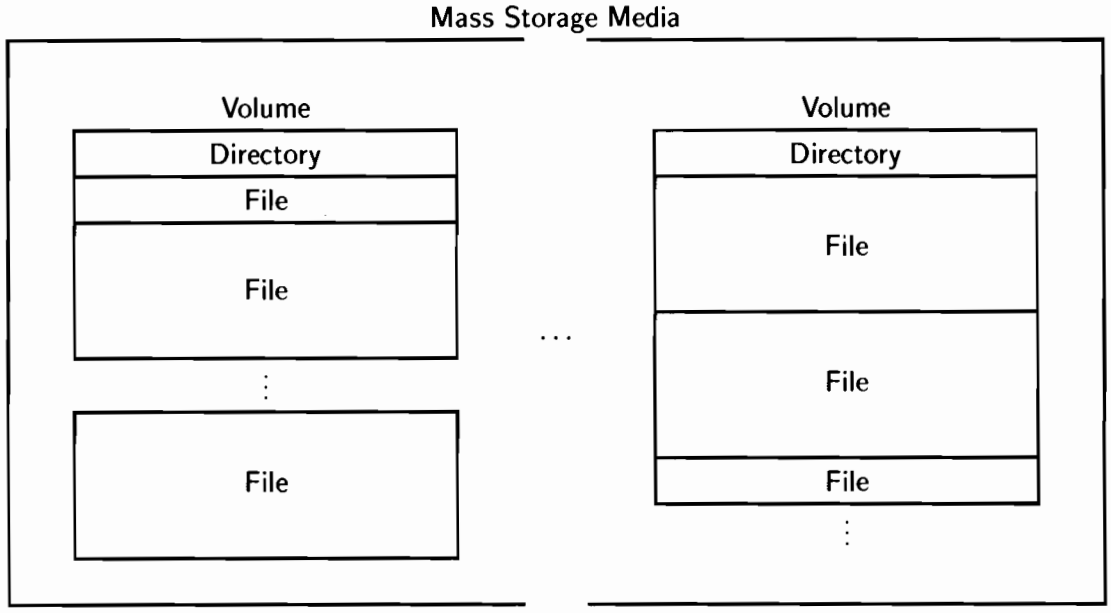


Structure of the Data Portion of a File

In a nutshell, mass storage access consists of the Pascal file system creating a *file* on a particular *volume*, and then writing to or reading from individual records in that file. Before showing examples of that, however, let's complete an overall picture of mass storage by looking at directories and volumes.

Mass Storage Organization (Non-Hierarchical Directories)

Mass storage is organized into volumes, each of which may contain several files. Here is a pictorial representation of the relationship between volumes and files.

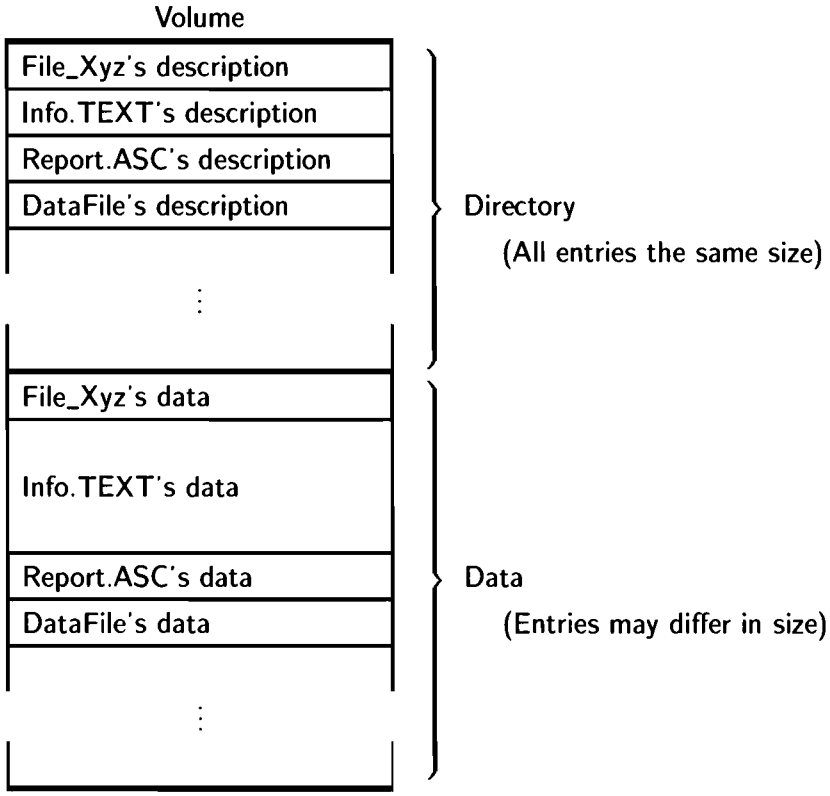


Mass Storage Device Organization

(Hard Discs, Flexible Discs, Cartridge Tapes, etc.)

Volume Structure

The term "volume" was chosen by analogy to a book. A book contains a table of contents and information. Similarly, volumes contain a directory or directories of the files in them, as well as the information in each file. Here is a graphic representation of how LIF volumes are organized.



Directory entries usually contain such information as:

- File name
- File type
- Start location of file (offset from beginning of volume)
- Number of blocks allocated to file
- Current length of file (in bytes)
- Date created
- Date last modified

However, the information contained in directory entries varies with the type of directory (e.g., LIF, WS1.0, SRM or HFS).

Classifications of Files

There are two main classes of files that the Pascal system can deal with:

- Item-oriented, or "fixed-record" files.
- Line-oriented, or "text" files.

Item-Oriented Files

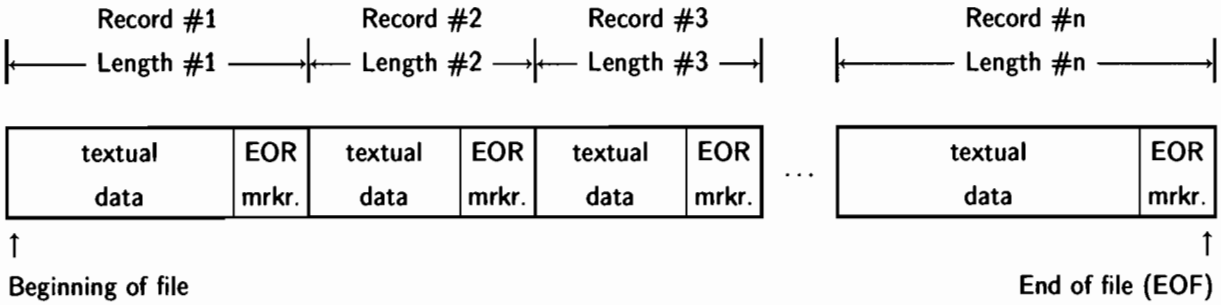
As the term "item-oriented" suggests, this type of file consists of data "items", each of which has a fixed size. Because of this, the name "fixed-record files" is often used. Records in these files can be any Pascal **type**, such as pre-defined simple types (e.g., **char**, **integer**) or structured types (e.g., **record**, **array**), and only data of that type can be put in the file.

Structure of Item-Oriented ("Fixed-Record") Files

When data is placed onto an item-oriented file, it is written in internal format; bit for bit, the same structure it occupies in memory.

Line-Oriented Files

As the term "line-oriented" implies, this type of file consists of records which are lines of text. Another common name for this type of file is "text files." Lines in such a file may vary in length (Length 1, Length 2, etc., in drawing below), so they are terminated by a unique "end-of-record" (EOR) marker. (Note that **.ASC** files have no explicit EOR mark; see "Other Types of Text Files" later in this chapter).



Structure of Line-Oriented "Text" Files

You can write or read either an entire record or part of a record at a time. And when reading, you can determine if you are at the end of a line or at the end of the file.

When data is placed onto an line-oriented file, it is *not* written in internal format; it is formatted to be readable by humans.

In a nutshell, the Pascal system locates a *file* on a particular *volume*, and then can write to or read from that *file*, one *record* or part of a record at a time.

Item-Oriented Files

In the previous section, a very brief mention was made of a "**file of** ⟨*type*⟩." Here we will delve further into what they are and why they are useful.

In past examples, text files were opened for writing with **rewrite**, and opened for reading with **reset**. Since text files are line-oriented, they were written to with **writeln** and read from with **readln** (although **write** and **read** could have been used in addition).

With item-oriented files, **writeln**, and **readln** won't work (the Compiler issues an error if **writeln** and **readln** are used). To open a file, you use an **open**, **append**, **rewrite** or **reset** statement. To write to and read from a file, you use **write** and **read**. Here is one advantage to item-oriented files: they can be *random-access* files (another term is *direct-access*). That is, to read or write item 54, you don't need to read or write items 1 through 53.

Creating and Writing to an Item-Oriented File

It is as simple to create a file of some type of data-type entries as it is to create text files. One slight difference, though, between **text** files and **file of** ⟨*type*⟩ files is that **file of** ⟨*type*⟩ files must have ⟨*type*⟩ defined by the user if it is not a standard Pascal **type**. The declaration **text** is a standard Pascal type, and so the user doesn't need to worry about defining it.

In a **file of** ⟨*type*⟩, ⟨*type*⟩ can be any valid Pascal **type** constructor, except those containing files; files cannot be nested. For example, you can use predefined data types to define the items for a file:

```
var
  FirstFile:        file of char;
  SecondFile:       file of integer;
  ThirdFile:        file of real;
```

You can also define your own data items with which to define a file:

```
type
  GI=               record
                      Name:         string[30];
                      Rank:         (Private, Lieutenant, Captain);
                      SerialNumber: string[13];
                    end;
  Squad=            array [1..11] of GI;
var
  GIfile:           file of GI;
  SquadFile:        file of Squad;
```

Write the following small program and execute it[1]:

[1] This and the following examples assume that you have a mass storage device at unit #3. If this is not the case for your system, modify the example appropriately for your hardware. (The Filer's Volumes command shows you the units that are on-line.)

```
program One;
var
  Test:          file of integer;
  I:             integer;
begin
rewrite(Test,'#3:DataFile');
for I:=1 to 10 do
  write(Test,I*I);
close(Test,'save');
end.
```

The variable **Test** is a file all of whose entries are of type **integer**. This means that whenever the program uses the variable **Test**, it is referring to a file which is accessed as integers being written to and read from it.

In the actual execution of the program, the **rewrite** statement causes a file called "**DataFile**" to be created on unit #3 (if it does not already exist). Since the file variable used in that statement is **Test**, and since the file variable **Test** is declared in the program to be of type **file of integer**, integers can be written, one at a time, to the file. The statement **write(test, I*I);** does just that.

Now enter the Filer and type

L :Return

You should get something like the following:

```
V3:                Directory type= LIF level 1
created 27-Feb-87 11.11.41 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test                       1         40 15-Mar-87
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=1 unused=1043 largest space=1043
```

The file size is 40 bytes, as indicated by the Filer. This is as it should be, since integers are four bytes (these are 32-bit integers) apiece, and we wrote 10 integers to the file. Note that there is no need for anything like the EOR markers (length headers or carriage returns) in this type of file, as there is in text files. The reason for this is that all the items in the file are the same size, so a simple multiply of the item number sought minus one (because the first item is record 1) by the size of the item gives the location of an item in the file. In text files, the lines can vary in length, so there needs to be some kind of delimiter separating them and random access is not possible.

Reading Sequentially From a File

Now that we've written something to the file, let's read it back. Enter this next program into your computer, then compile and run it.

```
program Two(output);
var
  Test:         file of integer;
  RecNumber:    integer;
  Value:        integer;
begin
reset(Test,'#3:DataFile');
for RecNumber:=1 to 10 do
  begin
  read(Test,Value);
  writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
  end;
close(Test);
end.
```

When the program runs, it reads sequential records from the file, and prints them to the screen.

Detecting the End of the File

If you do not know how many items the file contains, you can just keep reading until you reach the end, as below:

```
program Three(output);
var
  Test:         file of integer;
  RecNumber:    integer;
  Value:        integer;
begin
reset(Test,'#3:DataFile');
RecNumber:=0;
  repeat
  read(Test,Value);
  RecNumber:=RecNumber+1;
  writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
  until eof(Test);
close(Test);
end.
```

This program will read the file from start to finish. Record #10 is the last record, so when that is read, **eof(Test)** goes **true** and the program finishes. (Note that a **while not eof** loop is a bit better than a **repeat until eof** loop, because the **while** loop can also handle an empty file.)

Now consider the following program. It reads the file, but *in a user-specified order.*

```
program Four(input, output);
var
  Test:         file of integer;
  RecNumber:    integer;
  Value:        integer;
begin
open(Test,'#3:DataFile');
  repeat
  write('Record number to read: ');
  readln(RecNumber);
```

```
    seek(Test,RecNumber);
    read(Test,Value);
    writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
    until eof(Test);
  close(Test);
end.
```

The **eof** function still goes true when record #10 is read, even if it is the first one you read. This is probably not what you want. Thus, you will probably want to determine the exit-the-loop condition on something other than actual end-of-file, since the "end of processing" may occur anywhere.

Also note that the above program crashes if you specify a record number outside the range of 1 through 10. This is certainly not user-friendly. You can put in a check for this by using the function **maxpos**, which tells you what is the highest-numbered record in the data file (this does not work for text files).

```
program Five(input, output);
var
  Test:         file of integer;
  RecNumber:    integer;
  Value:        integer;
begin
open(Test,'#3:DataFile');
  repeat
  write('Record number to read: ');
  readln(RecNumber);
  if (RecNumber>=1) and (RecNumber<=maxpos(Test)) then
    begin
    seek(Test,RecNumber);
    read(Test,Value);
    writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
    end
  else
    writeln('Illegal record number specified.');
  until eof(Test);
close(Test);
end.
```

After being satisfied that you understand the above examples, remove any leftover files that remain from their execution from the disc now, so they won't clutter the Filer listings for future examples.

In the next section, line-oriented files, or *text* files, are discussed.

Line-Oriented (Text) Files

This section deals with files whose organization is oriented toward lines of text. These lines of text have different maximum lengths, depending on the file type:

`.TEXT`	1023 characters.
`.ASC`	32 767 characters.
`.UX`	No limit.
`Data`	No limit.

Note

The compiler will only process up to the first 110 characters on a line of a program source.

See the section "Other Types of Text Files," later in the chapter for more information.

Since the items in line-oriented files need not be the same length, some mechanism must exist whereby each line's length is determined at the time of reading or writing. This concept is different from *item-oriented* files, wherein all the items are the same structure *and* size. That type of file was discussed earlier in this chapter.

Creating a File

It is a simple thing to create a file on a mass storage device. Write the following small program and execute it:

```
program Six;
var
  Test:          text;
begin
rewrite(Test,'#3:Test2.TEXT');
close(Test,'save');
end.
```

The variable `Test`, as you can see from the declaration, is of type `text`. This means that whenever the program uses the variable `Test`, it is referring to a file which can be accessed as "lines" of text. The `rewrite` statement associates the variable `Test` with the file named `Test2.TEXT`. Note that if `Test2.TEXT` does not already exist, the file system creates it automatically. If `Test2.TEXT` already exists, all data is cleared out of it by `rewrite`.

In the actual execution of the program, the `rewrite` statement creates a file called "`Test2.TEXT`" on unit #3. Since the file variable used in that statement is `Test`, and since the file variable `Test` is of type `text` (as per the declaration part of the program), the computer knows that the file called `Test2.TEXT` can contain `text`. The program does not actually write any text to the file.

Files are sometimes used as temporary entities, existing only for the duration of the program. This is what the Pascal system assumes, unless you tell it otherwise. For this reason, the `close` statement is included in the program. The `close` tells the computer to "close" the file; that is, disassociate the actual file from the program currently executing, and the program disavows any knowledge of the file's actions or attributes. The file to be closed is specified by the first parameter. What is done with the file at its closure is specified by the second parameter; `'save'` in this case (note that `'save'` is a string parameter), which makes the file "permanent" on the mass storage device.

If the `close` statement doesn't have a second parameter, the file is closed with the same attributes as it had before the opening of the file with the `rewrite`. This file didn't exist before the `rewrite`; thus, if you closed the file without the second parameter, it would be gone after the `close`. Again, the `'save'` string[2] as the second parameter causes the file to remain in existence after the `close`.

Now enter the Filer and type

L : Return

which tells the computer to list ("L") the directory of the default (":") volume. You should get something like the following:

```
V3:                Directory type= LIF level 1
created 27-Feb-87 11.11.41 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test2.TEXT                0          0 15-Mar-87
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=0 unused=1044 largest space=1044
```

You may have more entries listed on your screen, but let's only deal with the `Test2.TEXT` file. Note that its name, size (in both blocks and bytes), and date of last change are listed. This confirms it: you *did* create a file with your program.

[2] The second parameter on the `close` statement may be uppercase or lowercase; letter case is completely ignored. The string `'lock'` does the same action as `'save'`.

Writing to a File

Go back into the Editor and modify your program. Place these two statements immediately after the `rewrite` statement:

```
writeln(Test,'Printing one line...');
writeln(Test,'...and another.');
```

This tells the program to write, to the file indicated by file variable `Test`, the two lines "`Printing one line...`" and "`...and another.`". Note that the single quote marks are merely delimiters of the string literals; they just specify where the text to be written starts and stops. They are not considered part of the strings, and thus are not written to the file.

Now compile and execute the new version of the program. After it finishes, enter the Filer again and look at the volume listing (do the L:Return command again).

```
V3:                Directory type= LIF level 1
created 27-Feb-87 11.11.41 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test2.TEXT              8       2048 15-Mar-87
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=8 unused=1036 largest space=1036
```

The file is a different size this time. This is not surprising, considering we put nothing in the file the first time, and thirty-five characters in it the second time. But why did the file size increase so dramatically? We put thirty-five characters into the file, and the file size increases by more than two *thousand* bytes?

The answer lies in the definition of `.TEXT` files. `.TEXT` files have these two characteristics:

- A file whose name at creation time ends in `.TEXT` contains a "header"—a 1K-byte area which does not contain text, but is an area for carrying information about the file. For example, when you create a `.TEXT` file with the Editor, margin information, markers, and all the other environment information are stored in this area.
- A `.TEXT` file takes up space only in increments of 1K bytes. Therefore, when we wrote the thirty-five bytes into the file, it took 1024 bytes. If we added more and more bytes to this file, its size would not increase until more than 1024 bytes (excluding the 1K-byte header) were written, at which time another 1024 bytes would be appropriated.

Reading a Text File with the Editor

Did the characters actually make it to the file the way we wanted them to?

To see if the two lines of text actually made it onto the file, let's try to read the file into the Editor and see if the text is there. When you enter the Editor, specify `Test.TEXT` as the file name to edit. Sure enough, the Editor displays:

```
Printing one line...
...and another.
```

This file can be edited in the usual way, re-stored, etc., just like a file which was created by the Editor. In fact, you cannot tell by looking at any particular `.TEXT`-type file with the Editor whether it was created programmatically, as above, or with the Editor.

Reading a Text File with a Program

Now that you've written something in the file, how do you read it back later? This is also an easy task. Enter this next program into your computer, then compile and run it.

```
program Seven(output);
var
  Test:         text;
  Line:         string[80];
begin
reset(Test,'#3:Test2.TEXT');
readln(Test,Line);
writeln(Line);
close(Test);
end.
```

When the program runs, it prints:

```
Printing one line...
```

It only read and printed the first line of the file; that's all we told it to do. To read and print the second line, merely add the following two lines right before the `close` statement:

```
readln(Test,Line);
writeln(Line);
```

With this modification, the program will read the first line from the file, print it, read the second line from the file, and print it. But now add a third pair of statements:

```
readln(Test,Line);
writeln(Line);
```

Now the program will attempt to read and print three lines from the file. But the file only contains two lines! What will happen when the program attempts to read the "third" one? Let's run the program and find out.

The computer displays the following (PC value may differ on your system):

```
Restart with debugger ?
Printing one line...
...and another.

-----------------------------------------------
error -10: tried to read or write past eof
PC value:    -1390528
```

What happened was this:

1. The computer successfully reads the first line of text from the file and prints it out.
2. The computer successfully reads the second line of text from the file and prints it out.
3. The computer tries to read the (nonexistent) third line of text from the file. However, there is no third line of text, so the statement is impossible to carry out. The computer considers this an error, and informs you by doing the following things:
    a. If your computer has a beeper, it beeps at you.
    b. A row of minus signs is printed to draw your attention to the error message to follow.
    c. The error message is printed. The "`error -10`" part of the message indicates that the problem was an I/O error. However, the message printed is not "`I/O error`"; the next part of the message tells you what kind of I/O error occurred: you `tried to read or write past eof`. The "`eof`" means "end of file".
    d. The PC (program counter) value is printed. This can be used for debugging, but we will not address that here.
    e. You get the option of restarting the program from the Debugger. The Debugger allows you to execute the program piece by piece, look at values of variables, etc. See the *Debugger* chapter of the *Pascal Workstation System* manual for more information on this.

Fine. Now we know what happened, but how can we stop it from happening again? And if we are trying to read a file someone else created, or even one that we created, we may not know how many records we'll have to read. There could be *any* number of records in a file (within the size limit of the physical volume).

Detecting the End of the File

The boolean function called `eof`, which tells you when the end of the file has been reached, also works with line-oriented files. Modify your second program so that it looks like the following:

```
program Seven(output);
var
  Test:         text;
  Line:         string[80];
begin
reset(Test,'#3:Test2.TEXT');
while not eof(Test) do        {new}
  begin                       {new}
  readln(Test,Line);
  writeln(Line);
  end;                        {new}
close(Test);
end.
```

The `while` statement checks every time through the loop (*before* executing the loop) whether or not the end of the file has been reached. The loop is only executed if the end of the file has *not* been reached. The following steps are executed:

1. Open the file.
2. The file is not a completely empty file, so `eof(Test)` is initially `false`.
3. Test for EOF. Read the first line ("`Printing one line...`") and print it. The end of the file has not been reached yet.
4. The loop iterates, since the end of the file was not found in step 3.
5. Test for EOF. Read the second line ("`...and another.`") The end of the file is found. Print the line.
6. The loop does not iterate again because the end of the file was found at the end of the second line of text.
7. Close the file.

Detecting the End of a Line

In addition to using the `eof` function, you can also use the function `eoln` in conjunction with text files. Obviously, an end-of-*line* function makes no sense in item-oriented files, which are not *line*-oriented.

If you read text from a text file into a string or a packed array of characters, the variable will be filled until either:

- An EOLN is reached
- The variable is filled to capacity

To read a line from a text file which is longer than the string you are putting it in, you can read the line in pieces, using `read`. When you get to the end of a line, use `readln` to go to the next line. Outside this loop, you use an EOF loop like before. Thus, you need an EOLN loop inside an EOF loop:

```
program Eight(output);
var
  Test:          text;
  Line:          string[4];
  I:             integer;
begin
reset(Test,'#3:Test2.TEXT');
while not eof(Test) do
  begin
  while not eoln(Test) do
    begin
    read(Test,Line);
    write(Line);
    for I:=1 to 100000 do;      {make a noticeable wait}
    end;
  readln(Test);
  writeln;
  end;
close(Test);
end.
```

When you run this program, notice that four-character (or less) pieces of the lines are read and printed. When the end of the line is reached, it is caught by the `eoln` function, and skipped over by the `readln(Test);` statement. Note that the `readln` is necessary; if it were not there, the program would stay at the same position in the file, reading empty strings into `Line` indefinitely.

Other Types of Text Files

In case there is a contradiction/ambiguity starting to loom in your mind, let us define more precisely the two definitions of the phrase "text file." Here are the two definitions:

1. The Pascal language's definition of "text file" is any file variable which is declared as type `text`, as opposed to `file of` ⟨*type*⟩, in the declaration section of a routine. This type of file can be written to with `writeln` statements, and read with `readln` statements, whereas `file of` ⟨*type*⟩ files cannot. Also, these files can deal with end-of-line conditions, and data is formatted into human-readable form before writing to the file.
2. Pascal's files which are declared `text` can be created as any of the following types (the rest of this section of the chapter elaborates on these various file types):
   - If a text file whose name ends with `.TEXT` is created, it will be what is called a TEXT file.
   - If a text file whose name ends with `.ASC` is created, it will be an ASCII file.
   - If a text file whose name ends with `.UX` is created, it will be an HP-UX compatible text file.
   - If a text file whose name ends with any other recognizable suffix (e.g. `.CODE`, `.SYSTEM`, `.BAD`) is created, an error will occur if you try to write to it.
   - If a textfile whose name ends with anything else is created, it will be a `Data` file.

Note that the file types are only determined by the file name *at the time of creation.* You can change the name of an existing file to anything you want, so conceivably, you could have an ASCII file called `Fred.TEXT`, or a TEXT-type file called `Data`, or a Data-type file called `TEXT.UX`.

In this section, we'll be exploring the different types of physical files, all of which are declared of type text in a Pascal program.

Creating ASCII and Data Files

Edit and compile program Six again. Modify the file-opening line such that it looks like this:

```
rewrite(Test,'#3:Test2.ASC');
```

Run the program again, and then modify the line again so that it looks like this:

```
rewrite(Test,'#3:Test2');
```

Run the program again, and then modify the line again so that it looks like this:

```
rewrite(Test,'#3:Test2.UX');
```

Run the program again, then enter the Filer and list the directory of unit #3: L :Return.

The Filer listing now looks like this (there may be other entries also, depending on what you've put on your disc):

```
V3:                Directory type= LIF level 1
created 27-Feb-87 11.11.41 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test2.TEXT              8       2048 15-Mar-87
Test2.ASC               1        256 15-Mar-87
Test2                   1         37 15-Mar-87
Test2.UX                1         37 15-Mar-87
FILES shown=3 allocated=3 unallocated=77
BLOCKS (256 bytes) used=10 unused=1034 largest space=1034
```

Note that although the same text was written to all three files, file sizes are usually different. The reason is that the other file types—ASCII, indicated by a .ASC suffix at creation, HP-UX compatible, indicated by a .UX suffix at creation, and Data, indicated by no suffix at creation—have their special characteristics, just as the .TEXT files, mentioned before.

TEXT-type Files

Files of type .TEXT have the following characteristics:

- A file whose name at creation time ends in .TEXT contains the "header" area for carrying information about the file.
- Line-endings are marked by carriage-returns.
- A .TEXT file takes up space only in increments of 1K bytes, and any unused space is zeroed.
- Logical end-of-file is specified by the first character in a line being chr(0).

Note that you cannot reliably put characters whose ord is less than 32 into a .TEXT file, because some of these characters are used in the file and have special meanings.

In our .TEXT file, the actual bytes placed in the file, excluding the header, are:

1. The characters "`Printing one line...`".
2. A carriage-return (`chr(13)`), indicating the end of a line of text.
3. The characters "`...and another.`".
4. Another carriage-return, indicating the end of another line of text.
5. ASCII nulls (`chr(0)`) for the remainder of the 1K-byte block.

ASCII Files

Files of type .ASC have the following characteristics:

- Lines are specified by two-byte length headers specifying actual length, and lines (in the file only) are padded to an even length. Any characters can be in the line (although the Editor, etc., may get upset with certain control characters).
- A .ASC file takes up space only in increments of 256 bytes on LIF discs. For non-LIF discs, the increment will be dependent upon the file system.
- Logical end-of-file is specified by a length header of −1, which is two consecutive bytes of value 255.

In our ASCII file, the actual bytes placed in the file are:

1. A two-byte (16-bit) length header, indicating the length of the upcoming line. Since our first line has 20 characters, the length header is 00000000 00010100, or an ASCII null, followed by an ASCII "DC4" character (`chr(20)`).
2. The characters "`Printing one line...`".
3. Another two-byte length header. Since our next line has 15 characters, the length header is 00000000 00001111, or an ASCII null (`chr(0)`), followed by an ASCII "shift-in" character (`chr(15)`).
4. The characters "`...and another.`".
5. A pad character (ASCII blank; `chr(32)`) to cause the next length header to start on an even byte.
6. Another "length header." However, since there are no more lines—the end of the file has been reached—there is a special flag value in this length header. Its value of −1 (two consecutive `chr(255)`s) tells the computer not to interpret the two bytes as a length header, but as an end-of-file marker.
7. ASCII nulls (`chr(0)`) for the remainder of the 256-byte block.

Data-type Files

Files of type Data have the following characteristics:

- Line-endings are specified by carriage-returns (chr(13)).
- A Data file takes up only the amount of space it needs, rounded up to the nearest block. That is, it is allocated in blocks, and its physical size remains an integer number of blocks. At file closure, however, the *logical* size is cut back to logical end-of-file, which can occur at any byte in the file.
- Logical end-of-file is specified in the directory.

In our Data file, the actual bytes placed in the file are:

1. The characters "Printing one line...".
2. A carriage-return (chr(13)), indicating the end of a line of text.
3. The characters "...and another.".
4. Another carriage-return, indicating the end of another line of text.

UX-type Files

Files of type UX have the following characteristics:

- Line-endings are specified by line-feeds (chr(10)).
- A UX file takes up only the amount of space it needs, rounded up to the nearest block. The block size is dependent on whether the file will be stored under a LIF, SRM, or HFS directory. In fact a UX file is just a special type of Data file, and therefore has almost identical characteristics. As with Data files, the computer does not know whether the file contains text or item-oriented data.
- Logical end-of-file is specified in the directory.

In our UX file, the actual bytes placed in the file are;

1. The characters "Printing one line...".
2. A line-feed (chr(10)), indicating the end of a line of text.
3. The characters "...and another.".
4. Another line-feed, indicating the end of another line of text.

There are intrinsic differences in these file types, and the Pascal operating system keeps track of them in ways other than just their names. As mentioned previously, you can change the name of an existing file to something that can be quite misleading, but the data format and other characteristics will remain the same.

To see some of the other ways that file types differ, go into the Filer and press [E] [:] [Return]. This makes an extended listing. Our volume will look like this:

```
V3:                Directory type= LIF level 1
created 27-Feb-87 11.11.41 block size=256
 Storage order
...file name....    # blks     # bytes    start blk ....last change... extension1
                type  t-code ..directory info... ....create date... extension2

Test2.TEXT               8        2048          12 15-Mar-87 15.49.38          0
                Text   -5570                                                    1
Test2.ASC                1         256          20 15-Mar-87 16.21.28          0
                Ascii      1                                                    1
Test2.UX                 1          37          21 15-Mar-87 16.21.41         37
                Hp-ux  -5813                                                    1
Test2                    8        7168           4 15-Mar-87 16.30.24          1
                Data   -5622                                                   37
< UNUSED >            1034                      22
FILES shown=3 allocated=3 unallocated=77
BLOCKS (256 bytes) used=10 unused=1034 largest space=1034
```

As you can see from the second column, the file type is noted elsewhere than just the name of the file.

Also note that the logical file size of the `Data` and `Hp-ux` files are indicated in the `extension 1` field on LIF discs.

In the next section, the rest of the file-manipulation routines are discussed.

More Details on Programming With Files

This section describes the operation of the Pascal file operations. It discusses the creation and disposition of files, and the basic operations on file data.

Creating New Files

A file is initially created by the `rewrite`, `open`, or `append` operations. However, `open` and `append` are usually applied to existing files.

These standard procedures each may take one, two or three parameters:

```
rewrite(⟨file_var⟩)
rewrite(⟨file_var⟩,⟨file_spec⟩)
rewrite(⟨file_var⟩,⟨file_spec⟩,⟨third_parameter⟩)
```

Here, ⟨*file_var*⟩ is the name of a Pascal file variable (for instance, a variable of type `text`, `file of integer`, etc.).

The "⟨*file_spec*⟩" parameter is the file specification. This parameter's type must be a string or a packed array of characters. The ⟨*file_spec*⟩ parameter may include volume specification (such as a volume name, unit number, HFS or SRM directory path) and size specification (such as `[100]` or `[*]`). Size specification is not supported in the `open` procedure.

The "⟨*third_parameter*⟩" is an optional parameter which is used to define the type of a file upon creation[1], and with Shared Resource Manager files to control shared access to the file. For access control on SRM systems, see subsequent sections of this chapter called *RESET, REWRITE, OPEN, and APPEND*, *SRM Concurrent File Access*, and *SRM Access Rights*. The file type is specified by entering either the suffix string or the numeric equivalent defining the type, inside "\" delimiters. For example, a file of type Hp-ux could be defined using the statement:

```
rewrite(f,'#11:MYDIR/myfile','\UX\');
```

or

```
rewrite(f,'#5:/USERS/hisfile','EXCLUSIVE\-5813\');
```

Where -5813 is the t-code as seen in the Filer's Extended listing.

Note that when combining SRM access and file type information within the third parameter, the file type specifier comes after the access rights specifier. The type specifier always uses the backslash, "\", as a delimiter. If the file type specifier exists as the "third-parameter", it overrides the suffix.

* [1] The use of the third-parameter to define file type upon creation is new to Pascal 3.2.

Temporary Files

We saw in one of the examples earlier in the chapter that when a new file is first created, it is considered "temporary," and it will remain so until it is closed with a specification that it be saved permanently. Such temporary files don't conflict with other files of the same name. A new file created by `rewrite`, `open`, or `append` will be thrown away when the program terminates *unless* the file is closed with `'LOCK'` or `'SAVE'` as the second parameter.

Size Specification Parameter

The allowable file name syntax depends on the Directory Access Method (DAM) being used; this subject is discussed in a section later in the chapter called *File Naming Conventions.* Current DAMs support LIF, WS1.0, SRM, and HFS file system access. However, all file names may have appended to them a specification of the size of the file, which the DAM may use at file creation time to allocate space. The size specification may take the following forms.

- Not present. The file will be allocated the largest available block of space for contiguous-file DAMs (LIF and Workstation 1.0 directory organizations), an indeterminate amount of space for the SRM or 0 for HFS. Example: `'CHARLIE.TEXT'`.
- `[*]` on the end of the file name. The file will be allocated the greater of these two quantities: 1) the second largest free block, or 2) half of the largest free block for contiguous-file DAMs (LIF and WS1.0); on the SRM, an indeterminate amount of space will be allocated. On HFS discs, 0 bytes will be allocated initially (but bytes will be allocated as needed). Example: `SUSANNAH[*]`
- `[`*n*`]` on end of file name, where *n* is a positive integer. The file will be allocated *n* blocks of 512 bytes each for contiguous-file DAMs and for HFS, or an indeterminate amount by the SRM. Example: `EXACTLY[1000]` is allocated 512 000 bytes.

Note

The Pascal `open` procedure does not support a size specification.

Anonymous Files

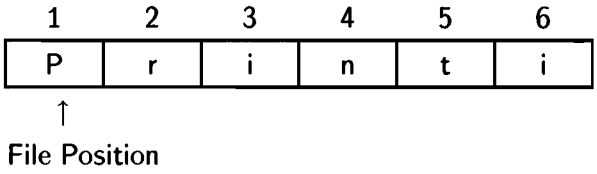
It is permissible to create anonymous files by creating a file without specifying a file name, for example `rewrite(F)`. Such files will always be placed on the system volume. Note however that there is no way to request a specific file size for an anonymous file; `rewrite(F,'[500]')` is not acceptable because there is no file name preceding the size specifier.

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

File Position

In order to understand the three modes a file can be in, we need to take some time to discuss the *file pointer* and the *file buffer*, denoted as the value pointed to by a file variable: `F^`.

A file pointer is associated with each open file. This pointer can be thought of as a marker indicating how much the file has been read or written. For example, the file pointer is initially pointing at the beginning of the file when the file is opened with **reset**, **rewrite**, or **open**. On the other hand, the file pointer is set to the end of the file if the file is opened with **append**. The file element pointed at by the file pointer is called the *current component*. Each time you read from a file, the current component is fetched. Each time you write to a file, the new information becomes the current component.



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

The Buffer Variable

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

File States

Every file which is open is in one of three states or modes at any given time depending on what was the most recent operation on that file. The file state has to do with whether you are reading or writing the file and whether you have referenced the buffer variable, `F^`. The three states are as follows:

- *Write* mode
- *Read* mode
- *Lookahead* mode

If the file is in write mode, `F^` has no special meaning other than as a variable, and referencing it causes no I/O to take place. This is the mode in which you normally assign to `F^`; for instance:

`F^`:=⟨*data item*⟩;

in preparation for a `put` statement. If you assign from `F^`; for instance:

⟨*data item*⟩:= `F^`;

in this mode you will get unpredictable results.

The `read` mode is also called the "lazy I/O" state, because in this mode the buffer variable refers to the current component of the file, but the File System does not fill it until the first time it is referenced. In this mode you normally assign from `F^` in order to read the next component of the file.

If the file is in `read` mode, referencing `F^` causes the current component to be fetched from the file and placed in the buffer variable. When this is done, the buffer variable is full and the file goes into the `lookahead` mode. Once the file is in the `lookahead` mode, `F^` may be referenced as many more times as desired but no more I/O will be done.

The `lookahead` mode is so called because we have "peeked" at the current component without having advanced completely past it. In actuality, the current component has been read into `F^` and the file pointer has advanced to the following component. However, the file system pretends that the current component hasn't been fetched yet. In this state the `position` function returns a value corresponding to the component in the file buffer, which is 1 less than that corresponding to the true file pointer. Also, in this state, `read(F,V)` will assign the value of `F^` to `V` instead of reading the next component of the file. On the other hand, if a write were done in this state, it would write the component at the true file pointer, and the `position` function would appear to advance by 2 instead of 1!

Use of the buffer variable (e.g. `f^`) is not required in Workstation Pascal. We suggest you do direct I/O from user variables, e.g. `write(f,myvar)`, unless you have existing code that uses the buffer variable. Direct I/O is usually faster.

Pascal Primitive File Operations

- The following operations put the file into WRITE Mode:

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

- The following operations put the file into READ Mode:

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

- The following operations put the file in LOOKAHEAD Mode:

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

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

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

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

OPEN(F) [with optional parameters]

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

If F was already open at the time of `open` and no file name is specified, the same physical file is referenced. If a file name is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit `close` is actually a `close(F,'normal')` and so the file will not necessarily be saved.

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

APPEND(F) [with optional parameters]

If F was already open at the time of `append` and no file name is specified, the same physical file is referenced. The procedure `append` positions to the end of the file and re-opens it for writing. If a file name is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit `close` is actually a `close(F,'normal')`, and so the file will not necessarily be saved. Any data written will get tacked onto the file; the original content remains valid.

If F is not already open and no file name is given, an anonymous file is created and the behavior is like the `rewrite` command.

If F is not open and a file name is given, `append` searches for an existing file of that name. If one is found, it positions to the end and prepares for writing; if none is found, it creates a new temporary file.

Restrictions on APPEND

Doing an `append` to text files is not allowed in the Series 200/300 Pascal Workstation implementation. It only works for Data files (`file of` ⟨*type*⟩).

If the file is in a volume with a WS1.0 or LIF directory organization, it may not be possible to `append`. For these directory types, `append` is *only* allowed if there happens to be free space on the disc *immediately following* the current end of the file.

Disposing of Files

A program terminates the association between a file variable and a physical file with the `close` procedure. For example, the call may specify that the file is to be deleted from the directory or made permanent. Here are some specific examples:

`close(F,'save');` `close(F,'lock');`	Both do the same thing; the file is made permanent in the volume directory. If the file is anonymous (has no name), then the file is closed and purged. Letter case is ignored.
`close(F);` `close(F,'normal');`	Both do the same thing. If the file is already permanent, it remains in the directory. If it is temporary, it is removed. Letter case is ignored.
`close(F,'purge');`	The file is removed from the directory whether or not it was permanent. Letter case is ignored.
`close(F,'crunch');`	The end-of-file (EOF) marker is set at the current file position; data beyond this position is lost. Otherwise like `'lock'`. Letter case is ignored.

Opening Existing Files

To open an existing file, you must give a file specification to the `open`, `append` or `reset` standard procedures.

RESET(F,'⟨*file_spec*⟩')
Opens an existing file for reading, and positions `F` to the beginning. If `F` was already open and no file name is specified, the file to be read is the one which was open. Otherwise, the file system searches for an existing file of the specified name and reports an error if none is found.

The statement `reset(F)` with no file specification will fail unless `F` is already open.

OPEN(F,'⟨*file_spec*⟩')
APPEND(F,'⟨*file_spec*⟩')
`open` and `append` search for the specified file. If one is found, then the association will be with that physical file. But note that if no file is found, then a new temporary file will be created (see the comments about file creation shown above).

Note that `open(F)` and `append(F)` without a file specification will create new files unless `F` was already open.

REWRITE(F,'⟨*file_spec*⟩')

When `rewrite` specifies the name of a file which already exists, a new temporary file is created. All output data goes to this new file instead of the old one. At the time the file is closed using `close(F,'lock')`, `close(F,'save')` or `close(F,'crunch')`, the old one is purged and the temporary file is renamed. Both `close(F,'normal')` and `close(F,'purge')` will purge the new file, leaving the old file intact. This prevents destruction of the old file in case the program terminates prematurely.

To get rid of the old file first, open it with `reset` and then do a `close(F,'purge')`.

Sequential File Operations

In Pascal there are two classes of files: `text`, or line-oriented file, and Data, or item-oriented files. Files of type `text` are so declared in the Pascal program:

```
var
  F:   text;
```

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

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

```
type
  Rec= record
         Name:           string[50];
         SocialSecurity: integer;
       end;
var
  SS:  file of Rec;
```

A `file of char` is not the same thing as a text file, because no lines are distinguished in the `file of char`, and only characters can be written to or read from them (not strings, etc.).

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

READ(F,V)

If `F` is open for reading (by `reset` or `open`), then this standard procedure will store into variable `V` the current component of `F` and advance to the next component. Note that `read(F,V1,V2,V3)` is equivalent to three `read`s in a row. In the `lookahead` mode, `read(F,V)` assigns the value of `F^` to `V` instead of fetching the next component of the file (i.e., no I/O is done).

WRITE(F,V)

If F is open for writing (by rewrite, append, or open), then the value of V is written as the current component of F, and F is advanced to the next component. write(F,V1,V2,V3) is allowed.

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

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

PUT(F)

The put and write operations are related. To output data using put, first store into the buffer variable the value to be written, then call put:

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

This sequence is equivalent to:

```
write(F,V);
```

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

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

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

The put(F) operation writes the buffer variable, F^, to the current component of the file.

GET(F)

This is the complementary operation to put, used for input. It throws away the current component value and advances the file to the next component.

In write mode, get changes the state of the file to read mode, but does not change the file position or do any I/O. For example:

```
open(F,'file_spec');  {puts file in write mode}
get(F);               {puts file in read mode}
V:=F^;                {fetches first file component into V}
```

In read mode, get causes one component to be fetched from the file, which advances the file position by 1, but that component is discarded. For example:

```
reset(F,'file_spec');  {puts file in read mode}
get(F);                {reads and discards one component}
V:=F^;                 {fetches second file component into V}
```

In `lookahead` mode, `get` discards the component in the file buffer, `F^`, and changes the state of the file to `read` mode. This causes the file position to reflect the true file pointer, thus appearing to advance it by 1. For example, this sequence of statements:

```
reset(F, 'filename');    {puts file in read mode}
V:=F^;                   {fetches first file component into V}
get(F);                  {discards F^, advances position}
```

is equivalent to this sequence:

```
reset(F,'filename');    {puts file in read mode}
read(F,V);              {fetches first file component into V
                              and advances file position}
```

Direct Access (Random Access) Files

Files of type Data (item-oriented files) may be accessed directly; that is, a program can specify that it wants to read or write the *n*th record in the file without scanning through the records in sequence. A file must be opened with the `open` procedure to allow direct access.

The components of a direct access file are numbered sequentially, with the first being number 1. (Note that there is no acknowledged standard in this area; for instance, UCSD Pascal numbers the first component of a direct access file as record 0. However, all HP Pascal implementations work as described herein.)

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

READDIR(F,N,V)

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

```
readdir(F,N,V1);
read(F,V2);
read(F,V3);
```

Also:

```
readdir(F,N,V);
```

is equivalent to:

```
seek(F, N);
read(F, N);
```

WRITEDIR(F,N,V)

The write-direct procedure positions `F^` to component *n* of the file, and then writes value `V`. Subsequent writes will place values in components *n*+1, *n*+2 and so on. For example:

```
writedir(F,N,V1,V2,V3);
```

is equivalent to:

```
writedir(F,N,V1);
write(F,V2);
write(F,V3);
```

Also:

```
writedir(F,N,V);
```

is equivalent to:

```
seek(F,N);
write(F,V);
```

SEEK(F,N)

As with the other direct-access procedures, file `F` must be opened (for both `read` and `write`). The procedure `seek` positions `F^` so that the next call to `read` or `write` will fetch or place component `N`.

```
open(F,'CHARLIE');
seek(F,100);
get(F);
V100:=F^;
```

This definition is certainly counter-intuitive in that the program *must not* do an initial `get` after opening the file, but *must* after the `seek` command.

The procedure `seek` works most smoothly (in the most natural fashion) if used with `read` and `write`:

```
seek(F,N);
read(F,V);
```

Remember that `seek` leaves the file in `write` mode, so that in order to read the current component by referencing `F^` you must first do a `get` command. That means that the following sequence:

```
seek(F,N);
write(F,V);
```

is the same as this sequence:

```
seek(F,N);
F^:=V;
put(F);
```

However, this sequence:

```
seek(F,N);
read(F,V);
```

is equivalent to the following sequence:

```
seek(F,N);
get(F);
V:=F^;
get(F);
```

POSITION(F)

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

Please be cautious with this function if the file is in the `lookahead` mode (i.e., if you have read the current component by referencing `F^`). In this mode, `position` is correct for reading, but it is 1 less that the correct value for writing.

MAXPOS(F)

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

Text Files INPUT and OUTPUT

A text file is composed of variable-length lines of characters. It differs from `file of char` in that the lines are separated by end-of-line marks. As mentioned at the beginning of the chapter, the Pascal Workstation File System supports four different text file representations. Text files are the basis of human-legible input and output. This means that they are used for "formatted" I/O, such as printouts.

Declaring a Text File

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

```
var
  F:          text;
```

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

```
program X(input,output);
```

However, they must *not* be declared in the body of the program.

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

```
program X(input,output,keyboard,listing);
var
  keyboard,listing:  text;
begin
 . . .
end.
```

Don't worry about why `input` and `output` must not be declared yet `keyboard` and `listing` must be; that's how it is. Note also that the four standard files are automatically opened by the Operating System before the program runs. The standard files do not generally appear in `reset` or `rewrite` statements, although they may be closed and re-opened if necessary. Closing and re-opening standard files is not recommended.

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

Representations of a Text File

The way lines of characters will be represented in a text file depends on the *file type*, which is determined when the file is originally created. The permissible file types are as follows:

- Text (suffix `.TEXT`)
- ASCII (suffix `.ASC`)
- Hp-ux (suffix `.UX`)
- Data (no suffix)

If the file name given in the `rewrite` statement which creates the file ends in the suffix `'.ASC'`, the file representation used is LIF (Logical Interchange Format) ASCII. In this representation, each line is preceded by a signed, 16-bit length field telling how many characters are in the line. In this representation, there is no restriction on what characters may appear in the line. (However, note that ASCII control characters will cause problems with the EDITOR subsystem.)

If the creation file name ends in the suffix `'.TEXT'`, the representation used is known as "Workstation 1.0" (or WS1.0) format. This format is compatible with the UCSD Pascal P-system textfile representation, and may be used as an non-HP interchange format.

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

The UX file type created when the filename ends in .ux or .UX at creation, is HP-UX text (and binary data) compatible. Any characters can be placed in the file; a line-feed (CHR(10)) denotes end-of-line.

The UX file type can be used whether or not UXTEXT_AM (found in INITLIB as shipped) is installed. If UXTEXT_AM is installed, TAB characters (CHR(9)) in a UX file will be automatically expanded to tabstops at every 8th position by blank filling if necessary. Note that TABs will only be expanded on input (not when writing to a file), only for UX files when UXTEXT_AM is installed. UX data files (`file of ...`) will not be affected. If UXTEXT_AM is not installed, TABs in UX text files will pass through uninterpreted.

If the text file is created anonymously (no file name given) or without a known suffix, the "Data" file representation is chosen. In this case, a carriage-return denotes end-of-line, and all other characters are passed through uninterpreted.

Note

If a file is to be used by the Editor, then you should not store control characters (characters with ordinal values less than 32) in it. These characters may cause erroneous cursor placement, which results in data being inserted or deleted in the file at the wrong place.

Note

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

The LIF ASCII representation can only be used if the LIF ASCII Access Method module (ASC_AM) is installed in your system's BOOT:INITLIB file. The WS1.0 format can only be used if the UCSD Text Access Method (TEXT_AM) module is installed in INITLIB. These modules are present in INITLIB when the Pascal system is shipped, but can be removed if not needed. The UX and data formats are "built-in" to the system.

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

Formatted Input and Output

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

HP Pascal supports two forms of character strings, generically referred to as PAC (for `packed array [1..n] of char`) and string. A PAC is a variable whose type specification is of the form

```
type
  T=   packed array [1..n] of char;
```

where *n* is some integer constant. The lower bound of a PAC subscript *must be 1* in HP Pascal, although Series 200/300 Workstation Pascal allows any arbitrary lower bound if the `$UCSD$` Compiler option is used.

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

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

```
type
  PAC=        packed array [1..10] of char;
var
  S:          PAC;
      .
      .
      .
S:='abcde';    {pad with 5 trailing blanks}
write(F,S);    {write 10 characters}
write(F,S:5);  {write first 5 chars}
write(F,S:15); {write 5 blanks, then all 10 chars of PAC}
```

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

```
type
  S=   string[n];
```

where *n* is a constant between 1 and 255 giving the maximum allowable length of the string. Strings differ from PACs in having an implicit variable "current" length. Usually the length of a string is the length of the last string value assigned to it, although string length can be explicitly manipulated by the standard procedure `setstrlen`.

When a string variable is read from a text file, its length is set to the length of the incoming string (see below).

Reading a STRING or PAC from a Text File

When a string is read from a text file, its length is usually determined by an end-of-line marker in the file.

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

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

```
while not eof(F) do
  begin
    repeat
    read(F,S);
    ⟨process s⟩
    until eoln(F);
  readln(F);
  ⟨any other desired processing⟩
  end;
```

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

RESET, REWRITE, OPEN, and APPEND

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

```
third_parameter   ::= [ shared access ] [ type specifier ]

shared_access     ::= [ concurrency_word ]
                  ::= [ password_list ]
                  ::= concurrency_word "," password_list

concurrency_word  ::= "SHARED"
                  ::= "EXCLUSIVE"
                  ::= "LOCKABLE"

password_list     ::= capability [ ";" capability ]

capability        ::= password ":" access_right_list

access_right_list ::= access_right { "," access_right }

access_right      ::= "READ"
                  ::= "WRITE"
                  ::= "PURGELINK"
                  ::= "CREATELINK"
```

```
                    ::= "SEARCH"
                    ::= "MANAGER"
                    ::= "ALL"

type_specifier      ::= "\" type_selector "\"

type_selector       ::= shortint | suffix
```

Note that `shortint = -32768..32767` and `suffix` is any recognized suffix without the leading period (e.g. `ASC`).

Note that in the passwords themselves, uppercase and lowercase letters are distinct. Examples of ⟨*shared_access*⟩ are as follows:

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

The above section concerning passwords and concurrency words applies only to SRM. HFS file systems have their own security methods, and LIF and WS1.0 do not support these security methods. See Chapter 3, The File System in the *Workstation System manual, Volume I* for more information.

As stated earlier, the third parameter is also used to define the type of a file on creation of the file. The permissible file types are already known, e.g. TEXT, ASCII, UX, CODE, etc. The type may be entered either in *suffix form* or in *numeric form* so that a file of type `TEXT` may be defined by "\TEXT\" or "\-5570\". The syntax permits SRM security and file types to be defined concurrently, providing the file type specifier comes after the *shared_access* specifier. For example:

```
rewrite(f,'#5:/USERS/MIKE/myfile','EXCLUSIVE\UX\');
```

Note that the file type specifier is always delimited with backslashes, "\", and that the suffix specifier may be in upper or lower letter case. The suffix specifiers are only recognized if the Access Method is loaded. For example, the `ASC_AM` must be loaded to recognize the `\ASC\` specifier. Leaving off a specifier indicates a Data type file. The `BAD`, `CODE`, `UX`, and `SYSTM` specifiers are always recognized as the Access Methods are always loaded.

The numeric file type specifier may be any number between -32768 and 32767 (except 3, which is reserved for directories and 0, which is reserved for HP-UX "special files"), however unrecognized numbers will cause the file to be formatted as type Data. The following are some useful numeric file type specifiers:

-5791	Files of type BDAT (Note: Only the Filer subsystem will do useful things with this file type. This is primarily an HP BASIC file type.)
-5622	Files of type Data
-5570	Files of type TEXT
-5813	Files of type HP-UX
1	Files of type ASCII

The Filer reports these numbers as the "t-code" field of an Extended listing.

Unrecognized *suffix form* file type specifiers will cause the file to be stored as type Data. The following are the currently recognized suffix file type specifiers for text files:

⟨*none*⟩	Files of type Data
TEXT	Files of type TEXT
UX	HP-UX compatible files. Note that this deviates from the name given within an extended directory listing (Hp-ux).
ASC	LIF ASCII files. Note that this deviates from the name given within an extended directory listing (Ascii).

SRM Concurrent File Access

Three modes of access to shared files are allowed:

EXCLUSIVE No concurrency. Only one workstation may open the file at one time. This is the default for all files opened on the SRM.

SHARED No controls. The file may be opened by any number of workstations for both reading and writing. This is particularly dangerous for multiple writers since, for performance reasons, some local buffering is done in each workstation. Different buffers may overlap parts of the same file, and may not contain identical data! Shared file users will not be aware of changes in actual end-of-file caused by the actions of other users.

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

The user-callable routines which support locking are provided in the library module called `lockmodule`, which is in the standard `LIBRARY` file (on the `SYSVOL:` disc). To use them, the program must `import lockmodule`. These specifications for these routines are as follows:

- `function lock (anyvar F: file): boolean;`

  This function returns `true` if the lock succeeded, or `false` if the lock failed because the file was already locked. Other I/O errors, such as `File not open`, generate an error condition which may be trapped by using `try/recover` (see the System Programming Language Extensions section of the *Pascal 3.0 Language Reference.*)

- `procedure waitforlock(anyvar F: file);`

  This procedure sends the SRM a request to lock the file, and then waits until it is confirmed.

- `procedure unlock(anyvar F: file);`

  This procedure releases the file so that another workstation can lock it.

File locking capabilities are primarily intended for data files (Pascal `file of` ⟨*type*⟩) which are opened for random access using the standard procedure `open`. Suppose that `F` is a file which is not already open. The cases are as follows:

- `open(F,'`⟨*file_spec*⟩`');`

  The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation. This is the default.

- `open(F,'`⟨*file_spec*⟩`','EXCLUSIVE');`

The existing file is opened for exclusive access. The **open** will fail if the file is already open by some other workstation. There are three ways to fix this, and they are presented in the order we suggest to attempt them: 1) Press [I] (for Initialize) from the main command prompt. This usually closes files opened by your workstation. 2) Rerun the configuration table program (**TABLE**). You can do this either by executing it like any other program ([X] from the main command level), or rebooting. 3) Shut down the workstation activity from the SRM console.

- `open(F,'⟨file_spec⟩','SHARED');`

  The file is opened for shared access. Any number of workstations may have the file open **SHARED** at the same time. They may read or write—there is no synchronization.

- `open(F,'⟨file_spec⟩','LOCKABLE');`

  The file is opened in such a way that no access is permitted unless the file is first put in the locked state. Any number of workstations may have a file open **lockable** at a time, but only one workstation may have the file locked.

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

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

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

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

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

SRM Access Rights

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

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

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

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

`SEARCH` capability is required on all directories along the directory path to a given file.

Write protecting a directory means that files cannot be added to or removed from that directory unless the correct `WRITE` password is given. This also applies to subdirectories immediately under the write protected directory. However, this does not prevent an existing file from being overwritten. For example, the Editor and Filer can successfully overwrite an existing file without the user providing a password.

The `reset` operation requires `READ` access to the file.

Both `READ` and `WRITE` capability are required if the file is opened by calls to `open` or `append`.

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

The `WRITE` capability on the directory in which it resides is required to close-with-`'purge'` a file, in addition to the `SEARCH` capability needed to open the file and `PURGELINK` capability on the file.

To close-with-`'lock'` a file, `WRITE` capability is required for the parent directory, in addition to the `SEARCH` capability needed to open the file.

If a password with `MANAGER` capability is used to open a file, any file operations may be performed, since the manager password would allow access types to be changed. For example, the following statement gives no public capabilities:

```
rewrite(F,'FILE1,'A:ALL');
```

While, the following statement keeps all capabilities except `MANAGER` public.

```
rewrite(F,'FILE1','M:MANAGER');
```

This second method allows any file operations to be performed, but the manager password `'M'` is required to change or set passwords.

HFS Permissions

HFS (Hierarchical File System) file access permissions recognize an **owner** of a file, the **group** of people who may have privileged access to a file, and all **other** users of the file. The file's **Mode** changes the access permissions of the file, i.e. it allows you to restrict or grant permission, providing you are the owner of the file.

Restriction Definition

By using a combination of user class and permission type, a total of nine permission states can be defined. You will only be in one user class for each file or directory, but each file has all of the permission types associated with it. This means that for a file we can combine the user class and permission type and display it in a shortened form as

```
  Owner        Group        Other
r  w  x      r  w  x      r  w  x
```

In a "directory info" column of an Extended directory listing, the code on the left ("644m") is the octal code defining the access rights. The first 6 specifies the rights of the Owner, the second 6 the rights of members of the group, and the 4 the access rights for all other people who are using the disc. The "m" signifies that 664 is the mode field.

The specification and modification of user class and permission types in terms of octal codes is covered in Chapter 5, The Filer, under the Hfs command.

Debugging Programs Which Use Files

The File System uses the `try/recover` and `escape` mechanisms (two System Programming extensions) in its normal internal operations. For instance, when opening a file, several escapes may occur internal to the File System or driver calls. However, these "errors" don't get passed on to the user program.

However, if the Debugger is used on such a program and error trapping is enabled, the Debugger will stop the computer on each internal escape. This behavior can be very confusing unless you understand what is happening. The telltale clue that this is happening is that the line number displayed by the Debugger (lower, right corner of the screen) doesn't change during the File System call.

The most common escape codes generated in this fashion are −10, 2080, and −26. You can suppress the Debugger's activity on these codes with the following "Escape Trap Not" Debugger command:

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

How Magnetic Discs Work

Now that the "theoretical" groundwork has been laid and we know how Pascal uses mass storage devices, how do they *really* work? How do bits stick to that little piece of plastic or aluminum?

Discs come in two types: "flexible" and "hard." Flexible discs are also known as "floppy discs" since they are light, thin, and can be bent slightly. Hard discs are sometimes called "fixed," since the disc is not removable from most hard disc drives.

Both types of discs work in essentially the same way. The disc is a platter similar to a phonograph record made of plastic or metal. The disc is coated with a smooth layer of microscopic, magnetizable particles similar to that used in recording tape. When the disc is in a disc drive, it spins very fast. As it spins, a magnetic sensor similar to the record/playback head in a tape recorder is held over the disc's surface. The disc drive has a mechanism used to move this head over various parts of the disc's surface.

The recording groove in a phonograph record is a continuous spiral from the outer edge to the middle. By contrast, magnetic discs are organized into a sequence of concentric but unconnected circular tracks. The computer must tell the disc drive where to place the head over a particular track in order to read or write data. The tracks themselves are logically broken up into blocks of data called sectors. Discs are often referred to as "blocked devices" because of this structure.

The smallest amount of data that can be read from or written to a disc is a single sector. The computer may read or write several sectors in immediate succession. Since the disc is spinning, the computer must usually wait until the desired sector rotates into position under the head once the recording head is positioned over the correct track. By processing one sector after another as fast as the disc is rotating, the time delay caused by waiting for the sector to get into the correct position can be effectively eliminated.

For various reasons, the computer may, after processing a sector, not be ready for the next one as it spins into position. By staggering the sectors on the disc it is possible to insure that the next logical sector rotates into place just when the computer is ready for it. This staggering technique is called *interleaving*, and it can greatly improve your system's performance. Using the wrong interleave factor can likewise drastically reduce your system's performance.

For example, imagine a track that has 16 sectors of data numbered 0 through 15. If the disc has an interleave factor of 1, the sectors are simply accessed in order of occurrence on the disc:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

After reading sector 0, the computer must immediately be ready for sector 1. If the computer isn't ready for sector 1, it will be missed and sectors 2 through 15 and 0 will pass under the head before sector 1 is again accessible. Thus, only one sector would be read on each disc rotation, not fifteen, which is highly inefficient.

Now suppose the computer's busy period after reading a sector is just a little less than the time that elapses while the next sector passes under the head. By placing sectors out of order on the disc as follows:

0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15

the computer can access sector 0, skip sector 8, access sector 1, skip 9, and so forth. It is not necessary to wait for an entire disc rotation between each pair of sectors. The numbering scheme shown is said to have interleave 2, since looking at every other sector accesses them in logical sequence.

The interleave factor for flexible discs is established by a process called initializing (some manufacturers use the term "formatting"), which must be done before the disc is used. Initializing is done by a utility program called `MEDIAINIT` supplied with your Pascal system. `MEDIAINIT` knows the appropriate interleave factor to use with various models of disc drives. The default interleave for the disc you are initializing is shown in one of `MEDIAINIT`'s prompts. This default is generally the best interleave factor for that particular device. For example, an HP 8290X defaults to an interleave factor of 3.

For hard discs, the interleave factor is established at the factory, and cannot be changed. Thus, initialization of hard discs serves mainly to find bad tracks and force the use of spare tracks, if necessary, not to change the disc's interleave.

No Room on Volume

Obviously there is a limited amount of space on a disc volume. When there is no room on a volume to create a new file, the system will report an I/O error.

If you are using WS1.0 or LIF discs, you may be able to solve this problem by using the Filer's Krunch command. This command consolidates all of the volume's free space by moving all of the files on a volume to the front of the volume. If you are on an SRM, contact the SRM administrator. If you are using an HFS disc, your disc is full.

Both the LIF and WS1.0 directory organizations are designed for "contiguous file space allocation". This means that when space is reserved for a file, the disc sectors set aside have sequential numbers. For instance a file requiring three sectors might get sectors 26, 27 and 28; or 31, 32 and 33. Files would *not* be allocated sectors 13, 56 and 2, because those sectors are not logically adjacent. To go back to the analogy with file folders in a drawer, if you had a file too big for one folder you might put it in two or three folders; but you'd want store them next to each other, not in random places in the drawer.

When a file is purged, all of its sectors are again available for use by another file. As files are created and purged, the disc space usage will develop "holes" of free space between valid files. This is called "fragmentation". It's possible for a considerable amount of free space to exist in the volume, yet be unusable because it is in pieces too small to use. Since files tend to be small compared to the total space on a volume, this problem usually occurs when the volume has relatively little free space left.

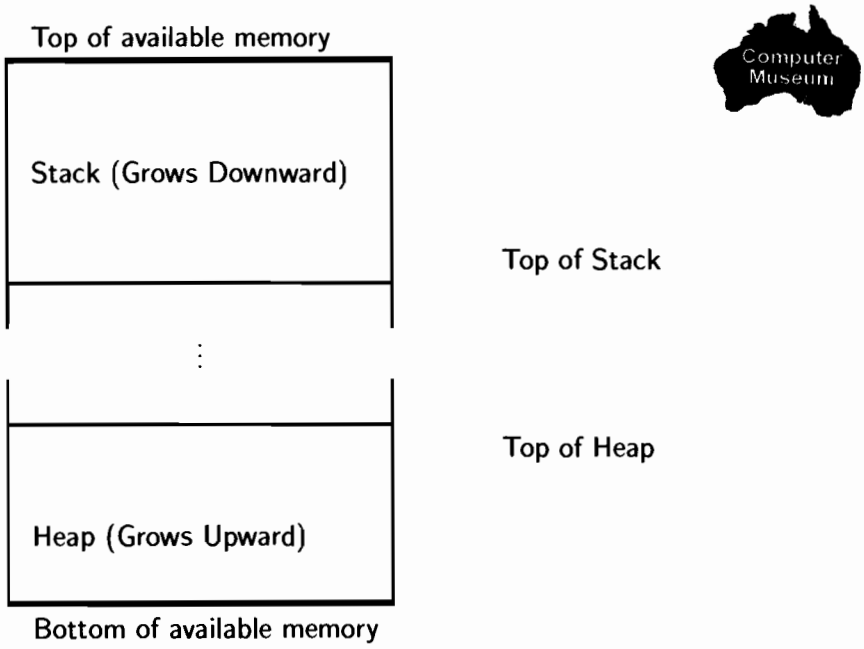
To see how fragmented a LIF or WS1.0 volume is, use the Filer's Extended Directory List command. This command lists both the files and the fragmented space on the volume. The listing will not show fragmented space for SRM or HFS volumes.

Dynamic Variables and Heap Management 16

Stack/Heap Architecture

The stack and the heap are two data structures inside your Pascal operating system which are used when procedures are called, variables are allocated, etc. The "heap" is the area of memory from which so-called dynamic variables are allocated by the standard procedure "**new**". It also contains currently loaded and p-loaded user and system program code. When a program begins running, aside from global space, it has available one area of memory for data. The program's stack begins at the high-address end of this area and grows downward; the heap begins at the low-address end and grows upward. If the stack and heap collide, a Stack Overflow error (**escapecode**=−2) is reported.

Conceptually, they look like this:



Dynamic Variables and Pointers

In more elementary Pascal programs, most variables are *static* variables (globals); that is, their storage space is allocated at the beginning of the program, and it remains allocated for the duration. This is adequate for many applications, but can cause problems at other times.

For example, when dealing with large arrays, often you do not know how big the array must be. When you run the program, the program may crash because the array is not big enough for this particular run. So, you increase the array size and, the next time, you get a memory overflow error; the machine does not have enough memory to allocate space for the entire array.

One way to deal with this problem is to let the program figure out—while it's running—how many elements it has to deal with. This means that the program allocates memory as processing takes place, and the memory used for one execution of the program is not necessarily the same as for another execution of the program.

Another example of where static variables are insufficient for the task is when your data items are very large—records or arrays of a kilobyte or more apiece—and you want to sort them. If you sort in such a way that you move the kilobyte-sized pieces around, the sort will take much longer than it needs to. The alternate method of just moving pointers is must faster for the machine to carry out, as pointers are four bytes apiece, regardless of the size of the object they point to.

Finally, there is a limit on total globals (including those used by the system) of 64K-bytes, and a limit of 32K-bytes of globals for any one program or module. By using heap storage, you can "get around" these limits.

Heap Management

Two disciplines are available for the recovery of the memory used by heap variables after they become unwanted: the **new/dispose** method, and the **mark/release** method. The first is more general; the second is simpler and faster.

MARK and RELEASE

This method uses two standard procedures to manage the heap in a purely stack-like fashion. The **mark** procedure is called to set a pointer to the next available byte at the top of the heap. Subsequent calls to **new** will all take space from above this point. When the program finishes with all the variables above the mark, **release** is called to move the top of the heap (the next available space) back to the value saved by **mark**.

```
program markrelease;
type
  ptr = ^ rec;
  rec = record
          f1,f2: integer;
        end;
var
  top,p: ptr;
  i: integer;
begin
  mark(top);        (* remember the base of the heap *)
  repeat
    for i := 1 to 5000 do
      begin
        new(p);     (* allocate from next highest heap address *)
        ...
      end;
    release(top);   (* cut back the heap; recover all space above the mark *)
  until false;      (* program will run forever *)
end.
```

When using this method, the computer does not prevent you from making the mistake of releasing to a point **above** the current top-of-heap! This could lead to corruption of programs or data. Thus, be careful not to release to a point above the current top-of-heap!

DISPOSE

Alternatively, the standard procedure **dispose** can be used to return each unwanted dynamic variable back to a pool of free space.

We saw the **new** function being used in the example programs earlier in the chapter, but in those applications, the data was in use until the end of the program, so there was no need for the selective removal of individual data items that **dispose** affords.

Calls to **dispose** will have no effect (the freed storage will not be reused) *unless* the main program and the modules containing the **new** and **dispose** calls are compiled with the Compiler option **$heap_dispose on$**.

```
program disposal;
type
  Ptr=   ^Rec;
  Rec=   record
           Next:   Ptr;
           F1, F2: integer;
         end;
var
  Top, P, Root: Ptr;
  I:            integer;
begin
mark(Top);            {remember the base of the heap}
repeat
  Root:=nil;
  for I:=1 to 5000 do
  begin
    new(P);           {after disposes, will allocate from free list}
    P^.Next:=Root;
    Root:=P;          {chain all cells together}
    ⟨do whatever other processing is desired⟩
  end;
  ⟨do whatever other processing is desired⟩
  repeat              {give back all cells one at a time}
    P:=Root;
    Root:=Root^.Next; {follow the chain}
    dispose(P);       {memory manager puts on a free list}
  until Root=nil;
until false;          {program will run forever}
end.
```

The recycling algorithm takes advantage of the fact that programs which use the heap operate on a great many variables of just a few types. Each type has a characteristic size. When a variable is disposed, it is saved at the front of a list of other variables of the same size. When a variable is allocated, the **new** routine first looks on the list corresponding to the size required; if there is a free object there, it can be allocated immediately. Usually there will be very little computational overhead for either **new** or **dispose**.

The memory manager maintains free lists for objects of sizes 6, 8, 10, through 32 bytes[1], and one more list for all larger objects. Objects are allocated from this last list on a first-fit basis. No dynamic variable is ever allocated an odd number of bytes.

[1] Prior to release 3.2, a free list of 4 byte objects was also maintained.

It is possible for the program to behave so that the heap becomes fragmented (broken into many small pieces). If a request then arrives to allocate space for a large variable, the memory manager will try to recombine the fragments to make a piece big enough to satisfy the request. The fragments will be sorted by address and adjacent ones merged.

The recombination process takes much longer than a simple allocation. Consequently, in real-time applications it is important to analyze the dynamic behavior of programs which use **dispose**.

Mixing DISPOSE and RELEASE

It is also possible to mix the regimes in a well-behaved manner. However, not all implementations of Pascal allow mixing these methods in a program. A program which does so may not run properly on other implementations.

If you **release** a properly **mark**ed pointer after some calls to **dispose**, the memory manager will leave on the free lists all disposed objects whose addresses are below the released location. All the space above the released location becomes free, whether or not it was disposed.

During this process the memory manager also recombines any adjacent free fragments, so **release** can also be used to reduce fragmentation. Just **mark** the current top of the heap, then immediately **release** to the same spot.

Note that the heap manager does not know about pointers within heap records, and cannot adjust them. For example, if you define:

```
list_element = record
                 value : integer;
                 link  : ^list_element;
               end;
```

If you create a linked list of heap elements of type **list_element**, disposing of one of those elements or releasing the heap to below any element may leave your list partly in unallocated memory space.

Error Trapping and Simulation 17

Introduction

The Systems Programming extensions to HP Series 200/300 Workstation Pascal have been provided to support error trapping and recovery. In order to use this mechanism, you will need to include the **$sysprog$** (or **$sysprog on$**) compiler directive at the beginning of the source program text. Note that these features may not be portable to other Pascal implementations.

Error Trapping and Simulation

The **try/recover** statement and the standard function **escapecode** have been added to the Pascal language to allow programmatic trapping of errors. The standard procedure **escape** has been added to allow the generation of soft (simulated) errors.

```
try
  ⟨statement⟩
  ⟨statement⟩
    . . .
  ⟨statement⟩
recover
  ⟨single, possibly compound, statement⟩
```

When **try** is executed, certain information about the state of the program is recorded in a marker called the *recover block*, which is pushed on the program's stack. The recover block includes the location of the corresponding **recover** statement, the current top of the program stack, and the location of the previous recover block if one is active. The address of the new recover block is saved, then the statements following **try** are executed in sequence. If none of them causes an error, the **recover** is reached, the recover block is popped off the stack, and skipped.

But if an error occurs, the stack is restored to the state indicated by the most recent recover block, and the recover block is popped off the stack. Files may be closed, and other cleanup takes place during this process. If the **try** was itself nested within another one, or within procedures called while a **try** was active, that previous recover-block becomes the active one. Then the statement following **recover** is executed. Thus, the nesting of **try** statements is *dynamic*, according to calling sequence, not statically structured like nonlocal **goto**s which can only reach labels declared in containing scopes. Be aware that if a local **goto** causes control to exit a try/recover block, a subsequent escape will cause the recover block and following code to be executed. This will not happen if the **goto** is non-local.

The recovery process does not "undo" the computational effects of statements executed between **try** and the error. The error simply aborts the computation, and the program continues with the statement following the **recover**.

When an error has been caught, the function `escapecode` can be called to get the number of the error. There are no parameters to `escapecode`. It returns an integer error number selected from the error code table (see the "Error Messages" appendix of the *Pascal Workstation System* manual). System error numbers are always negative.

Note that the `escapecode` can only be considered valid while executing a recover statement or block. If you check it outside a recover block, you may see a value that was set by some unrelated event.

The programmer can simulate errors by calling the standard procedure `escape(`*n*`)`, which sets the error code to *n* and starts the error sequence. By convention, programmed errors have numbers greater than zero. If an `escape` or error is not caught by a user recover-block within the program, it will be reported as an error by the operating system. Negative values are reported as standard system error messages, and positive values are reported as a halt code value. Note that `halt(`*n*`)` is exactly the same as `escape(`*n*`)`.

`Try/recover` statements are usually structured in the following fashion:

```
try
  ⟨some operation(s)⟩
recover
  if escapecode=⟨whatever you want to catch⟩ then
     ⟨recovery operation⟩
   else
     escape(escapecode);
```

This has the effect of ensuring that errors you *don't* want to handle get passed on out to the next recover-block, and possibly eventually all the way out to the system. All programs which are executed are first surrounded by the Command Interpreter with a `try/recover` sequence. The recovery action for the system is to display an error message.

The IORESULT Function

Normally the Compiler emits instructions after each file system I/O statement which verify that the transaction completed properly. If it fails, the program is terminated with an error report. The `escapecode` is -10 for file system I/O errors.

It is possible to trap file system I/O errors programmatically, using the `try/recover` statement. The System Programming function `ioresult` can then be called to discover what went wrong with the transaction.

Both the `escapecode` function and the `ioresult` function are needed for the following problem. Suppose, for example, you want to be able to enter values *of an enumerated type* into a program. This is easily done in HP Pascal, but if there is a misspelling, or an invalid token entered, the program bombs on "`error -10: bad input format`". How can this be avoided?

Put the **read** statement in the **try** section, and an error message in the **recover** section. If an error occurs during the read operation, **escapecode** and **ioresult** are checked. If they indicate that an illegal token was entered, print an appropriate error message, and ask for the same input again. Put the whole thing in a **repeat/until** loop so it continues until a correct answer is given.

```
$sysprog$
program TryRecover(input, output);
var
  Color:    (xxx, Red, Orange, Yellow, Green, Blue, Indigo, Violet);
  IOerror:  integer;
begin
  Color:=xxx;  {just a place holder, to see if a valid color was specified}
  repeat
    try
      writeln('Enter a color of the spectrum, then press RETURN or ENTER: ');
      readln(Color);
    recover
      begin
        IOerror:=ioresult;
        reset(input);              {clear bad data from input}
        if escapecode=-10 then
          if IOerror=14 {bad input format} then
              writeln('  (color invalid or misspelled)')
          else
            writeln('Escape code: ',escapecode:0,'  ioresult: ',IOerror:0)
                                   {some other I/O error}
        else
          if escapecode=-20 then   {stop key pressed}
            escape(escapecode)
          else
            writeln('Escape code: ',escapecode:0);
                                   {not I/O bad format or stop key}
      end;
  until Color<>xxx;
  writeln('You specified "',Color,'".');
end.
```

$IOCHECK$ and IORESULT

You may wish to intercept file system I/O errors (and other errors) programmatically rather than have them terminate the program. This can be done two different ways. The program or module must be compiled with the **$sysprog$** or **$ucsd$** Compiler option at the front of the source text. Both these options make available a system programming function called **ioresult**, which returns an integer value reporting on the success of the most recent I/O transaction. A result of zero indicates a successful transaction; other values are given in the "Error Messages" appendix.

Method 1

This method is the preferred one, and is the one used in the previous example. Compile the program or module with **$sysprog$** enabled, and use the **try/recover** statement to trap the errors.

```
$sysprog$
program TrapMethod1(input, output);
var
  Name:          string[80];
  F:             text;
  IOerror:       integer;
begin
  repeat
    write('Open what ".TEXT" file ? ');
    readln(Name);
    try
      reset(F,Name+'.TEXT');
      IOerror:=0;                  {If we get here, the RESET didn't fail.}
      writeln('  File successfully opened.');
    recover
      if escapecode=-10 then       {It's an I/O System error.}
        begin
          IOerror:=ioresult;       {Save it. (IORESULT is affected by WRITELN)}
          writeln('  Can''t open that file.  IOresult: ',IOerror:0);
        end
      else
        escape(escapecode);        {Pass non-I/O errors back to system.}
  until IOerror=0;                 {Keep trying until successful.}
end.
```

Method 2

This method is used in UCSD Pascal programs. In order for it to work properly, you must also suppress the I/O error checks normally emitted by the Compiler.

```
$ucsd$
program UCSD_TrapMethod(input,output);
var
  Name:          string[80];
  F:             text;
  IOerror:       integer;
begin
  repeat
    write('Open what ".TEXT" file ? ');
    readln(Name);
    $iocheck off$    {Disable file system error checking for next statement}
    reset(F,Name+'.TEXT');
    $iocheck on$      {Enable file system error checking}
    IOerror:=ioresult;   {Save it. (IORESULT affected by WRITELN)}
    if IOerror=0 then
      writeln('  File successfully opened.')
    else
      writeln('  Can''t open that file.  IOresult: ',IOerror:0);
  until IOerror=0;
end.
```

Note that **$iocheck off$** before the **reset** statement inhibits escape during the statement. However, **ioresult** will still be set correctly.

Be sure to turn **$iocheck$** back on when you have finished the I/O operation you are checking yourself. If you do not re-enable checking, later I/O errors may go unnoticed, leading to greater problems.

Extended Error Information

There are three types of run-time errors where your error-trapping will require the examination of extended error information. They are:

- File System I/O errors (**escapecode**=−10), and
- I/O library errors (**escapecode**=−26), and
- DGL (graphics) errors (**escapecode**=−27).

These are different than other, simpler, run-time errors, in that two values need to be checked in order to ascertain the specific error that occurred. This is different than, for example, an integer overflow error. In this case, **escapecode**=−4, and the error listings in the back of the *Workstation System Manual* states that −4 means "Integer overflow".

Get the "extended" error information in the following way. A "file not found" error causes an escape with **escapecode**=−10. However, an **escapecode** of −10 does not indicate *by itself* that some file was not found. The value of −10 only says, "Go look at **ioresult** for the rest of the definition of the error." Looking at the **ioresult** tells you that a file was not found. (Accessing the **ioresult** function requires either **$sysprog$** or **$ucsd$**.)

Be careful when trying to report an **ioresult** with a **write** or **writeln** statement since the write statement will "reset" the **ioresult**. For example:

```
try
  reset(F,'Nofile');
recover
  if escapecode = -10 then writeln('I/O error, ioresult=',ioresult);
  ...
```

The **ioresult** can never be reported as anything but 0, because **writeln** itself is an I/O operation, and before it executes, **ioresult** is reset to 0. A better approach is:

```
var
  ior : integer;
  .
  .
  .
try
  reset(F,'Nofile');
recover
  if escapecode = -10 then
    begin
      ior := ioresult;
      writeln('I/O error, ioresult=',ior);
    end;
  ...
```

Similarly, for I/O library errors, you need to check two different places. When you get an error **escapecode**=−26, all that tells you is that some I/O library error occurred. Now you need to check the I/O library's counterpart to I/O's **ioresult**, called **ioe_error**. (By the way, **ioe_error** and **graphicserror**, in the next section, are variables, unlike **ioresult**. Therefore, you do not need **$sysprog$** or **$ucsd$** to access them. Instead, import **iodeclarations, found in file LIB:IO or SYSVOL:IO, to access ioe_result**) The value of **ioe_error** tells you what kind of error occurred. In addition to this, there is a function called "**ioerror_message**" (imported from **iodeclarations**) which converts an integer to an appropriate error message:

```
writeln(ioerror_message(ioe_result));
```

Similarly, for graphics errors you need to check two different places. When you get an error **escapecode**=−27, all that tells you is that some graphics error occurred. Now you need to check the graphics counterpart to I/O's **ioresult**, called **graphicserror**. The value of **graphicserror** tells you what kind of graphics error occurred. To access **graphicserror**, which is really an integer function that does not require parameters, import **DGL_LIB**, which is found in GRAPH:GRAPHICS (or SYSVOL:GRAPHICS). If you have floating point hardware use FLTLIB:FGRAPHICS. If you have a coprocessor, FLT20:FGRAPH20 (or FLTLIB:FGRAPH20).

Determining a File's Existence

This section contains a program segment which is both a commonly needed capability and an instructive example. In many software packs, the user is allowed to store some kind of data, often specifying his own file names. Two things can happen at this point:

- A file by the specified name does not exist. Fine; create the file, store the data, and go on.
- A file by the specified name *does* exist. The computer should not automatically erase the old file and create the new one; there might be some valuable data lost. The computer should give the user the option of deleting the old file by that name, or specifying another name for the new file. At this point, another question must be asked; basically: "That file already exists; should I purge it?" If the user says yes, purge the file, create the new one, and go on. If the user says no, ask him for another file name.

Note that the second option above can happen repeatedly. That is, a user, upon being told that a file by that name already exists, can give another file name which already exists. Thus, the routine should repeat infinitely, if necessary, until a satisfactory file name is given.

```
$SYSPROG$
  ...
  var
    MyFile:       text;
    MyFileName:   string[80];
    Answer:       char;
    IOerror:      integer;
    NoFile:       boolean;
      .
      .
      .
    repeat
      write('File name to create: ');
      readln(MyFileName);
      NoFile:=true;
      try
        reset(MyFile, MyFileName);    {can't use REWRITE here}
        close(MyFile);
        repeat
          write('File "',MyFileName,'" already exists; shall I purge it? ');
          read(Answer);
          writeln;
          if not (Answer in ['n','N','y','Y']) then
            writeln('Please answer "Y" or "N".');
        until Answer in ['n','N','y','Y'];
        if Answer in ['y','Y'] then
          begin
            reset(MyFile,MyFileName);
            close(MyFile,'purge');
            writeln('Old file "',MyFileName,'" purged.');
            NoFile:=true;
          end
        else
          NoFile:=false;
      recover
        if (escapecode=-10) and (ioresult=10) then
          {do nothing; we've determined that the file is not found}
        else
          escape(escapecode);
        end;
    until NoFile;
  rewrite(MyFile,MyFileName);
  close(MyFile,'save');
  writeln('File "',MyFileName,'" created.');
  ⟨continue processing⟩
```

Note that **$sysprog$** must be specified in order to use **try/recover**.

Error Simulation

Here are two different facets to error simulation:

- Having your own set of errors, peculiar to a particular software package. For example, errors 1000 through 1050. These do not interfere or intermingle with any Pascal system errors, so when certain illegal operations in your software pack are attempted, you can cause one of your own errors to happen:

```
program MyProgram;
  . . .
⟨some error condition is detected⟩
halt(1000);
  . . .
```

or

```
$sysprog$
program MyProgram;
  . . .
⟨some error condition is detected⟩
escape(1000);
  . . .
```

(Again, **halt** and **escape** do the same thing.)

- The second facet of error simulation is: you don't *really* have an error, but you want the computer to temporarily think so, in order for it to take appropriate action. For example, suppose you set some conditions in the course of a program. If an error occurs during the condition-setting, you want to put things back in order. If an error doesn't occur, you want to do some processing, and *then* put things back in order. The point is: *either way*, you want to do the same return-to-normal code.

  Using **escape(0)**, you can cause a **recover** block to be entered, but the "error" number, 0, means "no error."

```
try
  ⟨attempt something which, if failure, goes to recover block⟩
  ⟨do processing⟩
  escape(0);  {cause control to go to the RECOVER block}
recover
  ⟨put things back in order⟩
```

  Note that the **escape(0)** causes control to enter the **recover** block in a nice, controlled manner.

Special Configurations 18

Introduction

The Workstation Pascal System is "self-configuring". As it boots, interface/device driver modules in the Initialization Library (BOOT:INITLIB or BOOT2:INITLIB) are loaded into memory and initialized. Then, the TABLE program determines what peripheral devices are connected to the computer (such as local and remote mass storage devices, printers, and so forth); if the driver module(s) for a particular interface or device is in memory, then the TABLE program can usually assign to it a logical unit number which makes it accessible to the File System.

The term "standard configuration" is defined to be any combination of computer and peripheral devices that will be configured by the Pascal system as it is shipped. This chapter describes how to change this "standard" system configuration.

Special note to existing users

The new (Revision 3.2) Hierarchical File System (HFS) capability **does not usually require changes** to CTABLE in order to support more than one hard disc. If HFS_DAM (the HFS driver) is in INITLIB, TABLE will automatically locate all HFS discs and assign one unit table entry to each disc. This may eliminate the need to modify CTABLE since more than one HFS disc can be automatically configured. If you wish to have more than one unit entry assigned to an HFS disc, see the "Extra HFS Unit Entries" section of this chapter.

Chapter Organization

This chapter contains many sections; however, they can essentially be split into three categories.

- A description of how the system boots and auto-configures itself.
- Brief descriptions of several possible configurations.
- Procedures for making changes to the "standard" configuration.

The System Booting Process

It will probably be beneficial to read about how the system boots and auto-configures itself, regardless of whether you want to change your system's configuration.

Example Special Configurations

Next, you will probably want to scan the possible "non-standard" ways that you can configure your system. The following sections briefly describe several common configurations:

- Hard Disc Partitioning
- Multiple On-line Systems
- Adding Interfaces and Peripherals
- Changing the System Printer
- Using Bubble and EPROM Cards
- Using Alternate DAMs (Directory Access Methods)
- Setting Up an SRM System

Modifying the Configuration

Then, when you know which configuration change(s) you want and which of the procedures you will need to use to make the changes, you can follow the procedures in the third major section of the chapter. These procedures are as follows:

- Coalescing logical volumes on hard discs into larger volumes (LIF only)
- Copying system files and changing their names
- Making an AUTOSTART or AUTOKEYS stream file
- Adding driver modules to INITLIB
- Modifying the auto-configuration program (CTABLE)
- Setting Up an SRM System

As an auto-configuration example, suppose you want to connect one HP9133V Hard Disc Drive and one HP7912 Disc Drive to your workstation. If these discs are not HFS volumes, the standard TABLE program assumes that it should assign 4 unit numbers to the 9133V hard-disc drive and 30 to the 7912. However, since it reserves only 30 unit numbers for *all* hard-disc volumes, the standard TABLE will not be able to access all 34 volumes (if that is the way that these discs have been or will be partitioned); it will either recognize all 4 volumes on the 9133V and only the first 26 on the 7912, or all 30 on the 7912 and none on the 9133V. Probably the easiest way to make all parts of both the above mentioned discs accessible is to use the hierarchical file system (HFS) on them. If you wish to use LIF, you would proceed as follows: first, "coalesce" the *last* 5 logical volumes on the 7912 into one larger volume (to change the total number of logical volumes to 26); second, set up the hardware so that the 9133V gets the lower unit numbers (11-14) and the 7912 gets higher numbers (15-40). Note that coalescing and multi-volume discs do not apply to Hierarchical File Systems (HFS).

An alternative way to make all parts of both discs accessible is to modify the standard TABLE program; the source program is called CTABLE.TEXT, and is supplied on the CONFIG: disc (ACCESS: disc for double-sided media).

A further way of making all parts of both discs accessible is to install HFS on both discs which will have the end effect of assigning only two unit numbers, one to each disc. With HFS it is still possible to divide a disc into thirty "sections" by creating thirty directories on that disc. In fact this can be more practical in many ways than 30 logical volumes (which have fixed sizes) on a single disc.

Descriptions of other configurations are given in the *Pascal User's Guide.* In one such example, the system files (such as EDITOR, FILER, and so forth) were copied to a hard disc (913x family). No file names were changed. An example AUTOSTART file was discussed in the same guide. It P-loaded some system files.

As an example of a "non-standard" configuration, suppose you want to use an HP98625 High-Speed Disc interface and an HP 98620 DMA Controller card with a CS80 disc drive. First, add module DISC_INTF to the INITLIB file (modules DMA and CS80 are in the INITLIB file supplied with your system). Then when the system is subsequently booted, the standard TABLE program will, barring other restrictions, automatically recognize the disc and make it accessible. (An alternative but less "permanent" way would be to eXecute module DISC_INTF after booting the system and then eXecute TABLE again). You will then probably want to copy most system files to the CS80 disc, which is another example of the second category.

Another "non-standard" configuration would be to use a SCSI bus interface card such as the HP98658A or HP98265A, and an HP98620B (or C) DMA controller to communicate with a SCSI disc drive. First add modules SCSIDVR and SCSIDISC to the INITLIB file (DMA is in the INITLIB file supplied with your system). Then when the system is booted, the standard TABLE program will, barring other restrictions, automatically recognize the SCSI disc and make it accessible. An alternative but less "permanent" method would be to "eXecute" the SCSIDVR and SCSIDISC modules after booting the system and then "eXecute" the TABLE program again.

As an example of a more difficult "non-standard" configuration, suppose you want to connect two HP7912 Disc Drives to your workstation, again assuming that these are of LIF or WS1.0 format. The standard TABLE program will not make both drives accessible, since it assumes that each disc needs to be allocated 30 unit numbers and assigns all 30 unit numbers available for hard discs to the 7912 with the highest priority. In order to access both drives with the File System, you will need to modify the standard TABLE program ("coalescing" will not work in this case). In this type of situation, you may want to change the default number of logical volumes that the system creates on each drive. After re-compiling and then running the properly modified program, the system will recognize and allow you to access all parts of each drive. You will probably want to replace the original TABLE program with the new version so that this configuration will automatically be made at the next power-up and system boot time.

The Booting Process

This section explains what is going on within the machine as the Pascal system is loaded and is intended to give you a few more insights into how the system works. It does not describe modules or how to boot or reboot your system. For more information on these topics, consult the following:

Booting the Pascal system	*Pascal 3.2 User's Guide*
Rebooting from the Debugger	*Pascal 3.2 Workstation System Volume 1*, the Debugger chapter
Rebooting from a program	*Pascal 3.2 Procedure Library*
Modules	*Pascal 3.2 Workstation System Volume 1*, the Compiler chapter

The Boot ROM

Inside the computer is a ROM (Read-Only Memory) that contains the information needed to begin loading an operating system. The loading process is often called "booting" because it is the computer's way of "pulling itself up by its own bootstraps." This ROM is therefore called the "Boot ROM". The Boot ROM is a non-volatile storage device; its contents are not lost when power is removed.

There are currently several different versions of the boot ROM. Thus, the booting process is slightly different depending on which version of boot ROM is in your computer. However, all perform the general steps outlined in this section.

When you power-up, the computer's central processing unit (CPU, which is a 68000-family processor) reads the first few bytes of this ROM, which begins at address 0. These bytes contain such information as the address of the first executable machine-language instruction and initial value of the stack pointer. After loading these values, the processor continues executing routines in the Boot ROM.

The processor next executes routines that perform a self-test and then displays the amount of memory installed in the computer. You may not see the amount of memory displayed if the CRT is just warming up. After self-test, the processor executes another Boot ROM routine that searches various mass storage devices (such as disc drives) for an operating system; the Boot ROM recognizes files of type **Systm** and with name beginning with the letters "**SYSTEM_**" (or "**SYS**" with Boot ROM 3.0 and later) as being operating systems. It also searches system ROM for ROM-based systems (such as BASIC).

Depending on its version and how many systems it finds, the Boot ROM will either choose a system or let you choose one (for instance, Boot ROM 3.0 and later versions allow you to choose one if you intervene in the boot process). With Pascal, this "SYSTM" type file is called "**SYSTEM_P**" and it will be discussed momentarily.

The Pascal System Discs

The Pascal system is delivered on either 5.25-inch (mini-floppy) or 3.5-inch (micro-floppy) flexible discs. The discs contain the operating system, subsystems like the Editor and Compiler, and several libraries and utility programs. The discs that you received are listed in the *Pascal User's Guide.* You will have to boot Pascal from these discs at each power-up unless you reconfigure your system. The disc called BOOT: (or BOOT2: for Pascal system supplied for the Series 300 computers) contains the SYSTEM_P file that will be loaded into memory first.

The System Boot File (SYSTEM_P)

The BOOT:SYSTEM_P program is an absolute-addressed program that contains the bare minimum Pascal operating system "kernel." (It was created using the Librarian's Boot command.) It is absolute-addressed so that the Boot ROM can use a simple loading routine.

The SYSTEM_P file consists of a linking loader (more elaborate than the loader found in the Boot ROM) and a few support routines. This kernel is loaded into volatile read/write memory (also called random-access memory, or RAM) from non-volatile memory (usually discs). The linking loader then loads the rest of the system.

In this system, there is no "kernel" in the closed sense of the term, such as a closed system like HP-UX. The system has an open design which allows modules to be added to the system — while the system is running. However, the term "kernel" will still be used in this text to describe the minimum working environment.

The loader then continues by completing construction of the operating system by loading the "Initialization Library" called INITLIB, which is also on the BOOT: disc.

The Initialization Library (INITLIB)

This BOOT: library file consists of modules that complete the kernel of the Pascal operating system. (Some of the modules are programs.) These modules mainly provide access methods (or device "drivers") for internal interfaces and peripheral devices.

The BOOT: disc contains an INITLIB designed for Series 200 computers and Series 300 computers using the 98546A Compatibility Display Card. The BOOT2: disc contains an INITLIB designed for Series 300 computers using their "native" bit-mapped displays. Other files on the two discs are identical.

Installing INITLIB Modules

As each INITLIB module is loaded into memory, it is bound to the operating system by a linking process. After the loading is complete, each program is executed once. The programs in INITLIB are referred to as "installation code;" their purpose is to properly initialize variables or allocate storage that will be used by these modules. Many interface-driver modules check to see if the interface they are to drive is there, and if not they don't install themselves.

Once INITLIB is loaded and the installation code has been executed, the system has found and identified all interface cards installed in the machine; however, no scan has been made for peripheral devices attached to those cards.

Auto-configuration of a peripheral device requires the device's driver(s) to be in memory at the time that the TABLE program is run (TABLE will be discussed in the next section). The HPIB module is an example of a driver for HP-IB interfaces. If the driver is part of the INITLIB file, then the device can be interrogated later, during the boot sequence, by TABLE (unless other conditions restrict). If the driver is not in INITLIB, then you must add it to the file (or alternately load the driver into memory by executing the installation program that contains the driver module).

Adding and Removing INITLIB Modules

Since the operating system is an "open kernel," you can add, replace, or delete modules within this library; more details regarding these operations are described in the Adding Modules to INITLIB section of this chapter. You must not change the order of modules in this library; neither should you link them together (with the Librarian), as that would result in rendering the programs non-executable.

Module LAST

The last piece of installation code in INITLIB is the program named LAST, which attempts to execute the BOOT: (or BOOT2:) disc files named STARTUP and TABLE. Here is the algorithm used to load and execute these two files; each file's function is described in a subsequent section.

1. If STARTUP is found on the "Boot volume" (i.e., the same volume on which the Systm file, such as SYSTEM_P, was found), then that program is loaded (but not executed).
2. LAST then looks for TABLE on the Boot volume. If TABLE is found there, then it is loaded and executed; it makes File System volumes accessible. If TABLE is not found, then only the keyboard, screen, and Boot volume will be accessible in very limited ways (see the brief description in the subsequent section called Failure of the TABLE Program).
3. If STARTUP was not found on the Boot volume, then the Boot ROM looks for it on the current system volume (at this point it might not be the Boot volume, because TABLE may have re-defined it).
4. STARTUP is then executed.

The Command Interpreter (STARTUP)

With the Pascal system delivered to you, the BOOT:STARTUP file is the Command Interpreter (or Main Command Level) program. However, you can write any program, optionally Link it with the Librarian, name it STARTUP, and with it replace the existing STARTUP file. It will then be loaded at power-up, instead of the Command Interpreter program.

If you use your own STARTUP program, be careful not to destroy the original STARTUP program. The recommended method is to use the Filer's Filecopy command to make a copy of the BOOT: (or BOOT2:) disc onto a blank initialized disc and then replace the STARTUP program with your new STARTUP program on that disc. Use the disc with the new STARTUP to boot the computer and your program will start running instead of the Pascal operating system.

The Auto-Configuration Program (TABLE)

The purpose of the TABLE program is to make devices accessible to the File System. Since this is one of the principle topics of this chapter, the subsequent section called Auto-Configuration is devoted to the intricate details of how this program works. For now, let's assume that it has already chosen the system volume and finish this overview of how the system boots.

The AUTOSTART and AUTOKEYS Stream Files

If present on the system volume and if data can be written on the volume (i.e., it is not a read-only volume), the AUTOSTART file is automatically streamed by the system at power-up; if the volume does not permit write operations (such as EPROM cards), then the AUTOKEYS file is streamed, if present. These files must be "stream" files, which are sequences of characters that are used by the system just as if they were commands typed from the keyboard (a "command stream"). Stream files are formally described in the Main Command Level Chapter of Volume I of this manual.

The AUTOSTART or AUTOKEYS file must be located on the volume designated as the system volume at the point that the TABLE program has finished execution. There is an AUTOSTART file on the BOOT: or BOOT2: disc. Here are the contents of the AUTOSTART file provided with your system.

```
1JAN70

xSWVOL
SYSVOL
3
wsSYSVOL:
qv
```

If you use the original boot disc on a single drive system, it is the AUTOSTART file which causes you to be instructed to place SYSVOL: in the drive and then press the [X] key. This AUTOSTART file then changes the system volume to "SYSVOL:". But because the boot disc was initially the system volume, the AUTOSTART file was found and executed. On dual-drive systems, the media in the second disc (nominally SYSVOL:) will usually become the system volume.

Libraries

The Pascal system is shipped with many library modules. Some are device drivers (in INITLIB or on the CONFIG: or LIB: disc for single-sided media and on the ACCESS: disc for double-sided media), while others provide procedures, etc. for applications such as device I/O and graphics (on the SYSVOL:, LIB:, and FLTLIB: discs). Once the system has booted successfully, you can use these libraries. INITLIB modules are described in the "Adding Modules to INITLIB" section of this chapter. Application libraries are fully discussed in the *Pascal Procedure Library* and *Pascal Graphics Techniques* manuals. You can also write your own libraries, as described in the description of Modules in the Compiler chapter.

The Auto-Configuration Process

A device is only accessible to the File System if it has been assigned a logical unit number. You may be familiar with the Filer's Volumes command, which shows the correspondence between logical unit numbers and volumes. Here is a typical display:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3   BOOT:
  4   SYSVOL:
  6   PRINTER:
```

The Unit Table

To make devices accessible to the File System, TABLE fills in entries of the Unit Table so as to correctly associate logical unit numbers with logical volumes (and the software required to access the devices on which those volumes exist). The Unit Table is actually a global system pointer variable called "Unitable," which points to a table that contains 50 entries — one for each logical unit (and potential volume). The Unit Table variable is accessed by many parts of the system, such as the Filer, Editor, and Compiler, when they want to use one of the devices.

This section describes how the standard TABLE program assigns Unit Table entries. To see the exact algorithms implemented in Pascal code, refer to the TABLE program called CTABLE.TEXT (found on the CONFIG: disc or the ACCESS: disc) and corresponding commentary later in this chapter.

Standard Auto-Configuration

The results of a typical auto-configuration process performed by the standard TABLE program are shown in the following table. Each entry is further discussed in subsequent text:

Standard Unit Table

Unit	Nominal Assignment
1	System CRT Screen (CONSOLE:)
2	System Keyboard (SYSTERM:)
3	1st priority floppy (drive 0, primary DAM)
4	1st priority floppy (drive 1, primary DAM)
5	Shared Resource Manager (remote mass storage)
6	System Printer (PRINTER:)
7	2nd priority floppy (drive 0, primary DAM)
8	2nd priority floppy (drive 1, primary DAM)
9	3rd priority floppy (drive 0, primary DAM)
10	3rd priority floppy (drive 1, primary DAM)
11-40	Hard discs (highest to lowest priority)
41	1st priority cartridge tape (LIF DAM)
42	2nd priority cartridge tape (LIF DAM)
43,44	1st priority floppy (same hardware as 3 & 4, but alternate DAM)
45	SRM system volume, if appropriate
46	HFS system volume, if appropriate
47,48	2nd priority floppy (alternate DAM for 7 and 8)
49,50	3rd priority floppy (alternate DAM for 9 and 10)

How Unit Numbers Are Assigned

In the Unit Table, certain unit numbers are preferentially assigned to particular classes of devices. Here are the general classes of devices:

- Unblocked devices (i.e., "byte stream" devices that do not have directories) like the keyboard, screen, and local printers
- Floppy disc drives (including 5.25-inch, 3.5-inch, and 8-inch)
- Hard disc drives
- SRM systems
- Cartridge tape drives

The floppy and hard disc drives and tape drives are all "blocked" devices.

Unblocked Devices

To fill the Unit Table, the TABLE program assumes that "unblocked" devices, such as the screen (CONSOLE:), the keyboard (SYSTERM:), and system printer (PRINTER:) are always present and assigns them to units #1, #2, and #6, respectively. However, it must scan for the presence of "blocked" devices (i.e., mass storage devices with directories). Once these devices are found, their locations (select code, HP-IB address, etc.) and attributes (type of disc drive, capacity, etc.) are put in the table entry corresponding to the logical unit number.

Blocked Devices

Here are the steps that the standard TABLE program goes through in assigning unit numbers to blocked devices.

Interfaces and Devices Scanned

In order to find mass storage (blocked) devices, the TABLE program first scans interface select codes 7, 8, and 14 for the presence of an HP-IB type interface: select code 7 is the built-in HP-IB interface; select code 8 is the factory default setting for optional HP-IB interfaces; 14 is the factory default setting for HP 98625 High-Speed Disc interfaces (fast HP-IB interface).

If an HP-IB interface is found, addresses 0 thru 7 are interrogated for the presence of blocked devices. (Most HP-IB peripherals identify themselves when asked politely.) The purpose of this interrogation is to determine what type of device (such as what family of disc drive, capacity of drive, etc.) is present at each location.

After scanning for an HP-IB interface, the TABLE program scans select codes 14, 15, and 28 for a SCSI type interface. Select code 14 is factory supplied for the HP98658A and the HP98265A interfaces. Select code 28 is the factory default setting for the 340 SPU. When using HP-IB and SCSI interfaces simultaneously, the TABLE program assumes that the SCSI interface card select code will be modified from 14 to 15.

Device Classes and Unit Numbers

The TABLE program makes a list of the devices found in each of these classes:

- Floppy discs — this class includes all 5.25-inch, 3.5-inch, and 8-inch floppy disc drives, all CS80 or SS80 devices that have a single *physical* volume with capacity less than 10 million bytes, and all SCSI removable media with disc capacity less than 10 million bytes.
- Hard discs — this class includes all supported 913x hard discs, all supported CS80 or SS80 devices that have a single *physical* volume with capacity greater than or equal to 10 million bytes, and all SCSI non-removable media.
- Tape drives — this class includes all supported cartridge tape drives, such as the HP 9144 Tape Drive as well as tape drives integrated into the CS80 Disc/Tape Drives.

Up to 10 devices can be on the list for each class. If more than 10 devices are found in a class, then only the *last* 10 found are maintained in the list.

As shown in the preceding Standard Unit Table diagram, groups of unit numbers have been reserved for each particular class of devices. For instance, unit numbers 3 and 4, and 7 through 10 are reserved for floppy discs. Unit numbers 11 through 40 are reserved for hard discs. Unit numbers 41 and 42 are reserved for tape drives.

Device Priority

The "priority" of a device is generally as follows: the later in the scanning sequence a device is found, the higher its priority is. Remember that HP-IB interfaces are scanned in the order of select codes 7, 8 and 14; and on each HP-IB interface, addresses 0 through 7 are interrogated. Thus, a device at 702 has higher priority than a device at 700 but lower priority than one at 800. However, if a device was used to boot the system, then that device may have the highest priority in its category.

The SCSI interface select codes are scanned after the HP-IB select codes. The prioritizing of SCSI devices is the same as for HP-IB devices. However, SCSI devices are the last to be found giving them higher priority than HP-IB devices.

Assigning Unit Numbers to Floppy Disc Drives

Units are assigned to floppy discs in pairs, according to device priority. For instance, if two dual-drive floppies are found, then the higher priority floppy device will be assigned unit numbers 3 and 4 and the lower priority device assigned unit numbers 7 and 8. However, if two single-drive floppy devices are found, then the higher priority device will be assigned unit 3 and the lower priority device assigned unit number 7. Up to three floppy drives (and thus pairs of floppy volumes) can be assigned unit numbers.

Assigning Unit Numbers to Hard Disc Volumes

Hard discs are also assigned unit numbers according to device priority; however, there is also another consideration. Since all hard discs currently supported by this system have capacities of several millions of bytes, the standard TABLE prefers to "partition" the *physical* volumes into smaller *logical* volumes. This partitioning is done on LIF format discs but not on HFS format discs. (Some hard discs are also organized to be accessed as separate physical volumes, rather than one large physical volume; see the subsequent Volume Sizes table for further information).

The following discussion applies only to LIF discs.

TABLE sets up Unit Table entries for hard discs according to two factors: the priority of each device, and the number of logical volumes it is assumed to have.

The logical partitioning of hard discs is made by the standard TABLE with the following algorithm. For each device on the list (of up to 10 devices), it calculates the number of volumes required by the device, assuming that the disc is now or will be partitioned; the default number of logical volumes assumed to be on each disc and the size of each volume depends on the size of the disc and configuration options. It then begins assigning unit numbers according to device priority; each device is assigned unit numbers according to the number of logical volumes *assumed* to be on the device, regardless of the number of volumes actually on that device. TABLE begins with 11, and continues either until all volumes have been assigned numbers or unit number 40 is reached, whichever occurs first.

At the point that it assigns unit numbers to a device, TABLE has *not* yet determined whether the disc has actually been partitioned. In fact, the disc may not have been initialized yet, or it may have been initialized but not partitioned as assumed. In the second stage of the assignment algorithm, TABLE looks on the disc for each volume's directory. Since these are logical volumes, each directory is assumed to be at an "offset" from the beginning of the disc.

If a valid directory is found at the expected location on the disc (i.e., at the assumed offset), then the corresponding unit number is assigned to the volume. For instance, if a valid directory is found in the first location, then it is assigned the first unit number for that disc (e.g., unit #11 will be assigned to the first directory on the highest priority hard disc device). As each subsequent directory is found, it is assigned the corresponding unit number. For example, if the only hard disc in a system is an HP 9133XV Hard Disc which has been partitioned and initialized according to the standard TABLE volume sizes for this disc, then it will be assigned 14 unit numbers (11-24).

If a subsequent directory is not found at its expected offset, then that area of the disc is assumed to be part of the last valid directory that preceded this one. For instance, if valid directories are found only in the 1st and 4th expected directory locations on an HP 9133V Hard Disc (assumed to have 4 volumes), then the first volume is assumed to be a coalition of the first three volumes (of the default size) on the disc.

If the first directory is the only valid one found, then the disc is assumed to be one single logical volume. For instance, if the only hard disc in a system is an HP 7911 Hard Disc which has been initialized and partitioned according to the standard TABLE volume sizes for this disc, then it will be assigned 27 unit numbers (11-37). However, if the disc was initialized by the Series 200/300 BASIC system (or coalesced into one volume using the procedure shown later in this chapter), then it will appear as one single, large volume and be assigned only unit number 11. In this case, the last 26 unit numbers allocated for the device (12-37) are *not* usable. If another hard disc (with lower priority) were added to this hypothetical system, then it would be assigned unit numbers beginning with 38, not 12.

Note

The only place that this logical partitioning information is kept is in the Unit Table entries for each volume; however, the information in the Unit Table is used by other parts of the system, such as the MEDIAINIT program that initializes (formats) the disc. The disc drive itself has *no* knowledge whatsoever of this partitioning scheme.

As another example of device priorities, suppose that you had an HP 9133XV drive and an HP 7908 drive in your system. Suppose also that the 9133 is at 702 and the 7908 is at 700. The 9133 is at the higher bus address (and is therefore found after the 7908 is found during the scan sequence), so it has the higher priority (assuming that the 7908 was not the boot device). The standard TABLE presumes that the 9133 is partitioned into 14 logical volumes, so it allocates 14 unit numbers (11-24) for the device. It then allocates 16 unit numbers (25-40) for the 7908 for analogous reasons.

As you might guess from the preceding discussion, even though there may be up to 10 devices in the list of hard discs, not all of the volumes they contain will necessarily be assigned unit numbers and thereby made accessible. Only the volumes to which unit numbers are assigned will be accessible. For instance, if the preceding example would have been two 7908 drives, then the highest priority device will be assigned 16 unit numbers (11-26), while the lower priority drive will only be assigned 14 unit numbers (27-40). If this disc had actually been partitioned into 16 logical volumes, then its last 2 volumes would *not* be accessible.

Note

If you plan to use your hard disc with BASIC, you should set up the disc as one logical volume under either LIF or HFS file systems. You will also want to be sure that all object code you wish to run is compatible with Pascal 3.2 (and works with HFS if you use HFS). See the File Interchange Between Pascal and BASIC section of the Technical Reference appendix.

Choosing the System Volume

The final step made by the standard TABLE is to choose the system volume. The operating system makes use of this volume for several purposes. For instance, after the system volume is designated at boot time, it is then inspected for system files (such as the EDITOR, FILER, and COMPILER). It is where the autostart file (AUTOSTART or AUTOKEYS) is assumed to be. It is also used by the system for storing temporary files that it creates for processes such as expanding Stream files. The system volume should remain on-line at all times if possible.

Here is the algorithm used by the standard TABLE program to determine which volume will be designated as the system volume; the "Boot volume" is the volume from which the BOOT: files named SYSTEM_P, INITLIB, and TABLE were loaded:

1. If a Boot volume was assigned a unit number *and* its capacity is greater than 300 000 bytes, then this device is designated as the system volume.
2. If step 1 did not designate a system volume, then search all volumes in the Unit Table. If a device with capacity greater than 300 000 bytes is found during this search, then it will be designated as the system volume.
3. If neither step 1 or 2 designated a system volume, then use the boot volume as the system volume.

Failure of the TABLE Program

By the way, if the TABLE auto-configuration program ever fails during the boot process, unit number 6 (normally the standard PRINTER: volume) is assigned to the screen, and unit number 3 is assigned to the "Boot device." You can only execute programs off of unit number 3 (with the Main Level eXecute command); it is otherwise inaccessible to the File System.

If TABLE is executed again but fails during this subsequent execution, then the Unit Table reverts back to its state before this unsuccessful try was attempted. (This is true *every* time that TABLE is subsequently executed.)

Example Special Configurations

This section describes several common types of special configurations. It outlines the general procedures required to implement them. The subsequent section called Modifying the Configuration provides the details of the procedures.

Hard Disc Partitioning

The way that the standard TABLE program prefers to partition hard discs was explained in the preceding discussion of Auto-Configuration. Here are the methods of changing this default partitioning.

- "Coalesce" adjacent LIF directories into one larger volume
- Modify the CTABLE program to partition the discs differently
- Use a disc which is of HFS format

The procedure for coalescing hard disc volumes is given in the subsequent Modifying the Configuration section.

Coalescing adjacent volumes will work well under these *general* circumstances:

- The disc is to be used with LIF directories.
- The total number of volumes that TABLE assumes it will find on *all* discs is less than 30.
- The sizes of volumes that you can make by coalescing an integral number of logical volumes is acceptable (i.e., the "resolution" of the default volume sizes is good enough).

If the desired configuration cannot be made by merely coalescing volumes, then you will have to modify the standard TABLE program (CTABLE.TEXT source file). Modifying the standard TABLE is also described in the Modifying the Configuration.

Note

If you are using HFS, neither coalescing nor modifying CTABLE is usually required. The system usually provides one unit entry per HFS disc automatically (it provides 2 units for the boot disc if it is an HFS hard disk).

Multiple On-Line Systems

If requested by operator intervention at power-up, computers equipped with Boot ROM 3.0 and later versions find all the on-line system Boot files (for example, SYSTEM_P) and display their names. You can choose the one you want to be booted. For instance, if you have a Pascal and a BASIC system on-line, you can choose which you want to boot.

Note

The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.

The term "BOOT: file" is used to identify a file used during the boot process; these files are on the BOOT: or BOOT2: disc shipped with your system.

By modifying certain BOOT: files (usually INITLIB and TABLE only) and uniquely renaming each different set, you can give yourself the option of choosing different Pascal system configurations at power-up. (This is not possible with the earlier Boot ROMs.)

For instance, suppose you want to have one system version that sets up SRM as the default volume and another that does not allow access to the SRM system. In such a case, you can create two systems, each of which is tuned for the desired usage; you can choose the one you want at power-up. To configure your system as such, you need to make duplicate copies of some system files and change some of their names. (You also need to set up the SRM system which is covered in detail in the SRM documentation.)

This type of configuration usually requires only the following type of modifications to the standard configuration:

- Change file names and copy them to different volumes
- Add module(s) to INITLIB

This type of change does not usually require changes to the TABLE program.

See the discussion of copying system files and changing their names in the "Modifying the Configuration" section for further details.

Adding Interfaces and Peripherals

Here is a brief summary of how to add several interfaces and peripheral devices to your system.

Hardware Configuration

You should configure each interface according to the instructions given in its installation manual. Most switches can be set to their factory defaults; however, the Pascal documentation will tell you when you will need to change the switch settings from the defaults.

Software Configuration

Using a peripheral device (and the corresponding interface) for File System operations may require this type of change to the standard configuration:

- Add module(s) to INITLIB

You may need to (or optionally want to) perform this type of configuration change:

- Modify the TABLE program

Here is a list of interfaces and peripheral devices and corresponding configuration modifications you will need to make in order to use each one. See the discussion of the type of change that you will make in the Modifying the Configuration section.

- **HP 98620 Direct Memory Access (DMA) Interface** The driver for this interface is module DMA, which is present in the original INITLIB. The interface is only used in conjunction with other cards. Note that the "DMA-CO" reported by the boot ROM in Models 330 and 350 is equivalent to the 98620, and uses the same driver.
- **HP 98622 GPIO (16-bit parallel) Interface** To drive this interface with the IO Library, add module GPIO. Performance may benefit from DMA hardware and driver. (GPIO is **not** required if you use the interface only for the HP 9885 disc; however, F9885 is required.) See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98624 HP-IB Interface** Using this HP-IB interface requires module HPIB, which is already present in supplied INITLIB. Performance may benefit from DMA hardware and driver. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98625 High-speed Disc Interface** This is a form of HP-IB interface, but it is only for use with discs. Modules DMA (already present in the standard INITLIB file) and DISC_INTF (not in the standard INITLIB) are required to use this interface, as is DMA hardware.
- **HP 98626 Serial RS-232 Interface** To drive this interface, install module RS232. DMA hardware will not improve performance. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98627 Color Output Interface** Using this interface is described in the *Pascal Graphics Techniques* manual. DMA hardware will not improve performance.
- **HP 98628 Data Communication Interface** To drive this interface, install module DATA_COMM. DMA hardware will not improve performance. See the *Pascal Procedure Library* manual for further details regarding I/O applications.

- **HP SRM Interface** Using this interface requires that you set up an SRM system. DMA hardware will not improve performance. See *Setting Up an SRM System* later in this chapter and the relevant SRM documentation, namely the *SRM System Manager's Guide* and *SRM Software Installation Manual* for further details.
- **HP 98630 Breadboard Card** This interface is intended for use only by system designers. DMA hardware may improve performance. See the *Pascal Systems Internal Documentation* for further details.
- **HP 98635 Floating-Point Math Card** No additional modules are required to use this card. However, you will need to use one of the FLOAT_HDW Compiler options; see the Compiler chapter for further details. The FLTLIB:FGRAPHICS module is optimized for use with this card and requires its presence. DMA hardware will not improve performance.
- **HP 98643 and built-in LAN Interfaces** Requires modules IOMPX and LANDVR to use this interface. DMA hardware will not improve performance. This interface is currently only usable with third party software products.
- **HP 98644 Serial RS-232 Interface** To drive this interface, install module RS232. This interface is very similar to the built-in RS-232 port found on most Series 300 computers. DMA hardware will not improve performance. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP built-in Parallel Interface** To drive this interface, install the PARALLEL module. DMA hardware may improve performance. See the *Pascal 3.2 Procedure Library* manual for further details regarding I/O applications.
- **Printers** The PRINTER module (already present in standard INITLIB) is required to drive all printers ("local" printers, not those on SRM), regardless of the type of interface being used. Additionally, the HPIB module (already present in INITLIB) is required for HP-IB printers. Printers with RS-232C interfaces can be used with the HP 98626 RS-232C Serial interfaces (or Series 300 built-in serial interfaces) if module RS232 is added to INITLIB. To drive a printer with an HP 98628 Datacomm interface, you will need to add the DATA_COMM module. Printers with parallel interfaces can be used with Series 300 built-in parallel interfaces if the PARALLEL module has been added to INITLIB. Usually DMA has little effect on performance.

  If a printer with an RS232C or Parallel interface is to be recognized by the File System (for example, volume PRINTER:), then you will need to modify the TABLE program. See the Changing the System Printer section for further details.
- **Graphics Output and Input Devices** To talk to "local" (i.e., non-SRM) HP plotters via HP-IB requires module HPIB (already present in the supplied INITLIB). Modules DATA_COMM and SRM are required if you are using the plotter spoolers on an SRM system. In addition, you will need to use modules in the GRAPHICS library. Normally, you will not access any local plotter through the File System (i.e., you will not access it through a logical unit number); thus, you will not need to add modules to INITLIB or modify the TABLE program. HP-HIL Graphical Input Devices do require the HPHIL module (either executed or added to INITLIB) and one of the drivers: MOUSE, DGL_ABS, or DGL_REL. See the *Pascal Graphics Techniques* manual for additional details. Usually DMA has little effect on the performance of graphic devices.

- **Mass Storage Devices** You will almost always access mass storage devices (such as disc and tape drives, EPROM, and Magnetic Bubble memory) from the File System. The TABLE auto-configuration program finds most "common" disc peripheral devices; however, to use "non-standard" devices like EPROM and Bubble cards, you will need to modify the program and add modules to INITLIB. See the corresponding sections of this chapter for further details. Many mass storage devices benefit in performance from using DMA hardware and driver module.
- **HP 98646A VMEbus Interface** The VMELIBRARY and IODECLARATIONS modules are required for driving this interface. DMA hardware will not improve performance. See the *Pascal 3.2 Procedure Library* for further details.
- **HP 98658A or HP 98265A SCSI Interface** To drive this interface the SCSIDVR module is required. DMA hardware and the DMA module will increase performance.

  To attach a SCSI disc, the SCSIDISC module is also required. For further information, refer to the subsequent section "SCSI Disc Considerations."

HP-IB Disc Performance Considerations

Disc performance is primarily determined by the device itself, but it may also be affected by the hardware used to interface the disc to the computer. Three common interface usages and their usual relative performance is given in the table below.

Lowest	Internal HP-IB or HP 98624 HP-IB interface (without a DMA card)
Higher	Internal HP-IB or HP 98624 HP-IB interface (with a DMA card)
Highest	HP 98625 High-Speed Disc interface and an HP 98620 DMA card (the 98625 card *requires* a DMA card)

The above table does not describe a hard-and-fast rule.

The 913xA Hard Discs Drives (excluding the V, B, and XV suffix drives) show an increase in performance when a DMA card is used.

Although it is not required, you should use an HP 98625 High-Speed Disc interface with CS80 discs for optimal performance. DMA hardware is required for using the HP 98625.

While the HP 9121, 9895, 9133 and 9134 discs can be used with the HP 98625 High-Speed Disc interface, they do not realize any increase in performance.

Note

Never use the HP 98625 High-Speed Disc Interface with an HP 82901, 82902, or 9135 disc drive.

SCSI Disc Considerations

The Pascal Workstation **does not** support all SCSI discs. It only supports HP SCSI discs. If you try to attach a SCSI disc that is **not** an HP SCSI disc it may or may not operate correctly.

The SCSI disc driver (SCSIDISC) expects attached discs to support the following SCSI commands as defined in the SCSI-1 standard:

- `TEST UNIT READY`
- `INQUIRY`
- `REQUEST SENSE`
- `READ CAPACITY`
- `READ EXTENDED`
- `WRITE EXTENDED`
- `FORMAT UNIT`

The SCSI disc driver also uses the following SCSI commands. Peripheral support of these commands is desirable, but not required.

- `MODE SENSE`
- `PREVENT/ALLOW MEDIUM REMOVAL`

The SCSI bus driver (SCSIDVR) expects attached devices to support the following SCSI messages as defined in the SCSI-1 standard:

- `IDENTIFY`
- `COMMAND COMPLETE`
- `ABORT`
- `INITIATOR DETECTED ERROR`

The SCSI bus driver also uses or recognizes the following SCSI messages. Note that peripheral support of these messages is desirable, but not required.

- `DISCONNECT`
- `SAVE DATA POINTER`
- `RESTORE POINTERS`
- `EXTENDED SYNCHRONOUS DATA TRANSFER REQUEST`

The Pascal Workstation SCSI disc and SCSI bus drivers were designed using the SCSI-1 standard. For more information, obtain the ANSI *Small Computer System Interface (SCSI)* manual, ANSI #X3.131-1986, from the American National Standards Institute, 1430 Broadway, New York, NY, 10018.

Changing the System Printer

Normally, the TABLE program assumes that the "system printer" (the PRINTER: volume, unit #6) is an HP-IB device at select code 7 with primary address 01. This section tells what is required to override this assumption. Here are the general changes you will need to make:

- If the printer is not an HP-IB device, you may need to add the corresponding driver module(s) to INITLIB. See the Adding Modules to INITLIB discussion in the subsequent Modifying the Configuration section.
- Modify the TABLE program so that it sets up the printer as the system printer (volume PRINTER:). See the Local Printer Type Option discussion in Modifying the TABLE Program.

Setting Up Printers with RS-232C Interfaces

The TABLE source program (CTABLE) provides a very clean way to set up an RS-232C printer as the PRINTER: volume. Here are the conditions required:

- The RS-232C interface can be an HP 98626 or 98644 RS-232C Serial or an HP 98628 Datacomm interface. The default select code is 9, but you can change the **dav** variable (device address vector) of the `local_RS232_printer_default_dav` constant in the CTABLE program to use another select code.
- In order to use either of the 98626 or 98644 interfaces, you will need to add module RS232 to INITLIB.
- In order to use the 98628 interface, you will need to add module DATA_COMM to INITLIB.
- The factory default settings for these interfaces are as follows:

  Interrupt level set for level 3

  Baud rate set for 2400 baud

  Stop bits switch set for 1 stop bit

  Bits/char. switch set for 8 bits

  Protocol set for XON/XOFF

  Parity set to off

If your printer uses other parameter(s), then set the interface card to match your printer. See the interface's installation manual for switch locations and settings. The HP 98644 RS-232 interface has no switches and must be configured programmatically (See its installation manual and the *Pascal Procedure Library* for more configuration information.)

In the CTABLE program, set the `local_printer_option` constant to **RS232**.

The select code of the interface is assumed to be 9; either set the interface to this select code or modify the **sc** parameter of the `local_RS232_printer_default_dav` in the CTABLE program to match the select code of your interface.

You may also need to change the `local_printer_timeout` constant to match your printer's characteristics. See the "Local Printer Options" section in the discussion of the CTABLE program.

Setting Up Printers with Parallel Interfaces

The TABLE source program (CTABLE) provides a way to set up a parallel printer as the PRINTER: volume. Before modifying CTABLE, the following prerequisites must be met:

- The parallel interface must be the systems built-in interface.
- The PARALLEL module must be added to INITLIB.

In CTABLE, set the `local_printer_option` constant to `PARALLEL`. It is also advisable to change the `local_printer_timeout` constant to match your printer's characteristics.

The select code for the parallel interface is assumed to be 23. If necessary, modify the `dav` variable (device address vector) of the `local_PARALLEL_printer_default_dav` constant in CTABLE to match the select code of your interface.

For more information, read the section "Local Printer Options" in the discussion of the CTABLE program form more details.

Using Bubble and EPROM Cards

Magnetic bubble memory and EPROM (erasable programmable read-only memory) are both types of non-volatile memory. The Pascal Workstation system allows you to use HP 98259 Magnetic Bubble Memory and HP 98255 EPROM cards as mass storage devices. This section briefly outlines what is required to configure your system to use these cards. The chapter *Non-Disc Mass Storage* gives further instructions.

You will normally be accessing these cards as mass storage devices. Here are the general steps required to make these devices accessible to the File System:

- Add the appropriate driver module(s) to INITLIB:

  To use a Bubble card, add the BUBBLE module to INITLIB. This module adds both read and write capabilities for Bubble cards to the system.

  To read EPROM cards (which have already been written), add the EPROMS module to INITLIB. (To program EPROMs requires an extra step, described shortly.)

- Modify the TABLE program so that it assigns a logical unit number to the device(s). See the discussion of Table Entry Assignment Templates in the Modifying the TABLE Program section.

See the the Non-Disc Mass Storage chapter for the complete description of using these cards.

Using Alternate DAMs

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

Pascal versions 2.0 through 3.12 support three mass storage directory organizations: the Workstation Pascal 1.0 format (WS1.0, similar to UCSD format); HP's Logical Interchange Format (LIF); and the Shared Resource Manager's (SRM) hierarchical, or "structured," directory format (SDF).

Pascal 3.2 and later versions support four mass storage directory organizations: WS1.0, LIF, SRM, and the Workstation Hierarchical File System (HFS).

In the case of Shared Resource Management discs, the DAM is supported in the SRM itself; what Pascal supports is the communication of DAM requests to the SRM. The SRM method can only be used with remote mass storage over an SRM hookup. The other three methods can be used with almost any local mass storage device.

The DAM used for each logical unit is selected by the TABLE configuration program. The standard TABLE selects LIF as the "primary" DAM, and, for pre-3.2 versions, UCSD (Workstation 1.0 compatible) as "secondary". For 3.2 and later versions, HFS is the "secondary" DAM. (Sometimes the word "alternate" is used instead of "secondary".) The primary DAM is the one used for *blocked* units in the range of #1 through #40 (except #5, auto-configured as the SRM unit). The secondary DAM is used by blocked units in the range of #43 through #50 (with these exceptions: #45 is auto-configured as the SRM system unit, if appropriate; #46 is auto-configured as the HFS system unit, if appropriate; memory volumes are always created using the primary DAM).

The secondary DAM is available to allow discs with the secondary directory format to be used by Pascal programs and the Filer utility. The secondary DAM is in no way restricted from normal use by the File System; discs in the secondary DAM units can be read and written directly by Pascal programs.

If Pascal 3.2 is booted up, just as it was shipped, the primary DAM will be LIF. The secondary DAM will be HFS, but the HFS_DAM module is not in INITLIB. You must add the HFS_DAM module (found on the HFS: disc or the HFS1: disc) to INITLIB before HFS discs will be recognized. To make the Pascal 1.0 (WS1.0) format the primary DAM, you will need to change the TABLE program. See the section called Modifying the Configuration Table later in this chapter for further details.

CAUTION

Do not maintain both LIF and UCSD directories on one logical volume. The directories have no knowledge of each other's existence, so one can readily destroy data in the other.

Comparison of LIF and HFS DAMs

HFS is a hierarchical file system that can be used on local discs. LIF file systems only permit a single level of directory.

With HFS, only one unit in the unit table is usually needed per disc, whereas LIF may occupy many units depending on the type of disc in use.

Pascal, BASIC, and HP-UX implementations of HFS are entirely compatible. All of Pascal's directories and files on HFS are always available to BASIC, unless permissions have been deliberately set to disallow such access. However, Pascal allows a LIF hard disc to be divided into "soft volumes". In this case, only the first soft volume is accessible to BASIC. Any remaining soft volumes on the disc are inaccessible to BASIC (and any other operating system which supports LIF).

For LIF file systems, all the space allocated to a single file must be contiguous. This is not the case for the HFS file systems. Consequently, if the free disc space is fragmented, the LIF DAM may be unable to create a new file of a specified size even though there is enough total free space. However, HFS has a different method of storing files and keeping track of where they are on the disc, so it is possible to store a single file in several parts at different, non-contiguous positions on the disc.

From the logic discussed above, it is clear that in a LIF file system it may not be possible to extend (or append to) an existing file even if the disc volume has some free space. The free space must be immediately following the existing file for it to be possible.

Letter case is significant for file names in both file systems as implemented on the Pascal and BASIC Workstations. HFS file names may be up to 14 characters long. LIF names are restricted to 10 characters.

The HFS DAM uses a cache. A cache is a part of the main memory of the computer where copies of disc blocks are kept to avoid accessing the disc more than is necessary. It is possible to alter the characteristics of this cache to suit your configuration and level of usage of HFS discs. See the section on User-configurable HFS Parameters at the end of this section. The HFS DAM caches only "control information" such as directories, indirection blocks, inodes, etc. User data is not cached.

The file access times are greater for HFS discs than for LIF discs. Since HFS has a higher "overhead" than LIF, an HFS disc will have less usable space than a LIF disc of the same size. HFS does have the benefits of compatibility with HP-UX and the ability to restrict access to specified users.

Comparison of LIF and WS1.0 DAMs

WS1.0 DAM is a DAM which has been supplied with Pascal since revision 1.0.

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

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

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

Workstation 1.0 file names may be up to 15 characters long. LIF names are restricted to 10 characters. In many cases this difference need not be a problem. Most file names used by the Pascal system end in a five-character suffix such as ".TEXT" and ".CODE"; hence the useful part of such names is 10 or fewer characters. The LIF DAM implementation encodes recognized standard suffixes into the suffix of a LIF file name, so that nine characters are available for the significant part of the name. This encoding is transparent when using the Pascal Workstation, as is the decoding back into the full suffix when necessary. When viewed from another operating system, the encoding is not transparent.

Recommendations For Selecting Primary DAM

If you are a new user and have no existing discs in the WS1.0 format, *we recommend that you use the system as supplied, with LIF as the primary DAM and HFS as the secondary DAM.* LIF is an HP standard for information interchange among computer systems. For instance, the BASIC and HPL systems that run on your Series 200/300 computer use LIF directories. The boot device must have a LIF or HFS directory unless you are booting from SRM. If you want to use HFS, you do not need to change the supplied TABLE program. You can leave LIF as the primary DAM and still have access to your HFS disc(s).

Note that the HFS DAM module is not in INITLIB as shipped; it is the file `HFS_DAM` on the HFS: disc (or HFS1: disc).

If Pascal, versions 2.0 through 3.12, is booted as shipped, the primary DAM will be LIF. The secondary DAM will be WS1.0. HFS is not compatible with versions prior to 3.2. We suggest you take all HFS discs off-line when booting versions of Pascal prior to 3.2. This prevents the earlier versions from "misunderstanding" HFS discs.

If Pascal 3.2 or later version is booted as shipped, the primary DAM will be LIF. The secondary DAM will be HFS, however the WS1.0 DAM is still available on the CONFIG: disc. To use this DAM you will need to modify the TABLE program and install the DAM.

If you have discs generated by Pascal 1.0 you have two choices. Both require a change in CTABLE. Change the constant "`thisversion`" to "`ucsdversion`" and then:

- Adopt LIF for new volumes but access your Pascal 1.0 files and directories through the limited number of alternate DAM units.
- Transfer your old files on Pascal 1.0 volumes to new LIF or HFS volumes.

The choice is primarily one of convenience, although in the long run there are advantages to LIF or HFS since these are "HP Standard" file systems. Since Pascal programs which ran under the 1.0 release must be recompiled to run under Pascal 3.0 and later versions, you should convert your user discs as well.

Moving Files Between WS1.0 and LIF Volumes.

The following steps outline the method of moving files from one directory type to another. For 3.2 and later versions, it is necessary to modify the standard TABLE program such that WS1.0 is the secondary DAM. See the *Modifying the Configuration* section later in this chapter for more details. Execute the WS1.0DAM, run the modified TABLE, then:

1. Put the ACCESS: disc in a disc drive and press [F] to run the Filer.
2. Put the source disc in a drive configured for its type of DAM, and the destination disc in a drive configured for its DAM. (See below)
3. Use the Filer's Filecopy command to move files from one disc to the other. The Filer commands will work with either DAM.

Note that the alternate DAM units allow either type of file system to be used in the same drive. For instance, you can copy a file from #43 to #3, both of which are assigned to the right-hand flexible disc drive in a Model 236 or 226 computer. For example:

Press [F] (for Filecopy), then enter:

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

The Filer will tell you to when to swap discs.

Remember that the name of a file in a WS1.0 directory may be too long for a LIF directory. You may have to invent a shorter name.

By the way, it's a good idea to develop the habit of using uppercase letters in the names of LIF files, because some other systems will not allow or recognize file names with lowercase letters.

Note that directories created by Pascal 1.0 have either 77 or 233 entries, whereas WS1.0 directories created by Pascal 2.0 and later have a variable number of entries specified by the user. Thus you can use Pascal 2.0 and later versions to create WS1.0 format directories which aren't readable by Pascal 1.0; whereas all Pascal 1.0 directories are readable by Pascal 2.0 and later versions, providing the WS1.0 DAM is installed.

User-configurable HFS Parameters

As discussed earlier in the section *Comparison of LIF and HFS DAMs* there is a method of tuning the HFS DAM to suit your configuration and HFS disc usage. This method involves the use of a module which must be inserted into INITLIB along with the module HFS_DAM. The source of this module is provided on the HFS: or HFS2: disc and is called HFS_USER.TEXT. Below is a listing of HFS_USER.TEXT.

```
{
{ User-configurable HFS parameters.
}
module hfs_user;

export

type user_rec = record
    user_cache_bytes: integer;
    simultaneous_hfs_discs: integer;
end;

const cache_info = user_rec[
    user_cache_bytes:  15*1024,
    simultaneous_hfs_discs: 1
];

implement

end.
```

As can be seen from the listing, the default size of the cache is 15K-bytes and the default number of `simultaneous_hfs_discs` is 1. These are the two parameters which you can alter to tune the HFS DAM.

Let us consider the first parameter. This parameter specifies the size of the cache memory in bytes. This is a section of RAM, reserved during booting, which is used by the HFS DAM to improve efficiency. No user data is stored in the cache memory, only organizational and control information. Although there is no specific maximum size for this parameter, a value above 30K-bytes will not give you any great advantage for typical use of the disc. Size should be specified in multiples of 1024 bytes; if you specify less than 10K-bytes, the cache will automatically be set to 10K-bytes.

The second parameter can be increased to represent the number of HFS discs you are likely to use *simultaneously* (not the number of units on a single disc). This is not simply the number of discs you have attached to your system. *Simultaneous* means,for example, that you have a program which has files open on a number of HFS discs at the same time. It does not mean that you use a number of discs during the day to copy files from one to the other. Increasing the value of `simultaneous_hfs_discs` to a number greater than one without concurrently increasing the `user_cache_bytes` will generally result in a degradation of performance as opposed to the desired increase. It is recommended that when increasing the value of `simultaneous_hfs_discs`, the `user_cache_bytes` be increased by the number of discs multiplied by the size of an HFS superblock. The exact size of a superblock depends on the disc in use but, as a guide, 3 to 4 K bytes is typical.

An example where this *tuning* would be advisable is if you had a stream file that compiled programs where the source existed on one disc, the object code went to another disc, and the listing to possibly a third disc.

Note that HFS cache memory is not recoverable as user memory except by rebooting.

Modifying the Configuration

This section describes the mechanics of modifying the "standard" configuration of the system as it was shipped to you. Here are some possible configuration changes:

- Coalescing adjacent hard disc volumes
- Copying system files and changing their names
- Using AUTOSTART and AUTOKEYS Stream files
- Adding driver modules to INITLIB
- Modifying the standard TABLE program
- Setting Up An SRM System

Coalescing Hard Disc Volumes (LIF Only)

As discussed previously, you can manually coalesce adjacent logical volumes on hard discs. For instance, suppose that you have an HP 9133V hard disc drive which is soft partitioned into the standard 4 logical volumes; the standard volume size is approximately 1 Megabyte. However, you want to increase the size of the first logical volume. You can easily coalesce the second volume with the first to double the size of the first. (This type of change is *only* possible with 913x Option 10 discs; 913x discs without this option cannot be logically partitioned.)

Overview of the Example Procedure

To coalesce the two logical volumes in this example, here are the steps you will take. Note that all existing files in both volumes will be destroyed.

1. If the disc has not been initialized, then you will need to do so before continuing with this procedure.
2. Invalidate the directory of the second volume by overwriting it with the data in a file. (Since the purpose of this step is to invalidate the directory, this file must *not* resemble a directory.)
3. Change the Unit Table by running the standard TABLE program. TABLE will find the invalid second directory, invalidate the corresponding Unit Table entry, and enlarge the volume size parameter of the preceding Unit Table entry (the first volume's Unit Table entry). This step sets up the Unit Table in preparation for coalescing the two volumes. Note that the first volume's directory *on the disc* has *not* been changed at this point; it is still the original size.
4. Create a new directory on the disc for the first volume; this directory will reflect its new size. To do this, you will need to destroy the first directory on the disc and then use the Filer's Zero command to zero it. The Zero command will read the size for the first volume *from the Unit Table*, since it will not have found a valid directory on the disc. The two volumes will then be "coalesced" *on the disc* when the first directory is enlarged as it is zeroed.

Note

When volumes are coalesced, the unit numbers formerly used by the coalesced volumes are not "freed up." For example, if volumes 12 through 14 are coalesced into volume 11 (as shown in the example that follows in this section) these 3 unit numbers are still allocated to the original partitions of the disk. The next unit number allocated to a hard disc volume will be 15 in the example.

Prerequisites

You should perform this operation *before* placing any valuable data in the volumes to be coalesced; however, if you have already used the volume, then you can back-up these files on another volume (such as a floppy disc or another hard disc drive). Once a volume has been coalesced with another, any data in it *cannot* be accessed.

The Example

The following procedure is an example of coalescing the second volume of an HP 9133V hard disc with the first volume, which results in approximately a 2-Megabyte first volume; the original third and fourth volumes will be left at the default size of approximately 1 Megabyte.

1. If the disc to be partitioned (here, the 9133V) has not already been installed with switches set properly, do so now. Set the drive's HP-IB address to a value that will ensure that it has a high enough priority to be assigned unit numbers. (Device Priority is fully discussed in The Booting Process at the beginning of this chapter.) For this example, we will assume that it will be assigned unit numbers 11 through 14.

2. If the disc has not already been assigned unit numbers (such as during a previous boot sequence), then use the eXecute command to run the standard TABLE program. If this program is not currently on an on-line volume or P-loaded into memory, then you will need to insert the BOOT: disc into a drive. Press [X] at the Main Command Level. The system prompts with this question:

   ```
   Execute what file?
   ```

   Enter the file specification of TABLE; the following specification indicates that it is on the BOOT: volume:

   ```
   BOOT:TABLE.
   ```

   The trailing period is required to suppress the otherwise automatic ".CODE" suffix, since this file's name on the disc has no suffix.

   When TABLE has finished, the disc should have been found and assigned unit numbers (we will assume 11 through 14). However, if it has not been previously initialized and directories zeroed, then unit numbers assigned to it will not show up in a Filer's Volumes command (they are invalid because the corresponding directories were found to be invalid).

3. At this point there are two situations possible: the disc either has or has not been initialized.

   a. If it has *not* been initialized, do so now; proceed with step 4.

b. If it has already been initialized, you have two more alternatives.

   If volumes have been coalesced and you want to split them (or if it is a disc initialized as one large volume), then you will need to first partition it into smaller volumes. Proceed with step 5.

   If it is has already been initialized but is still partitioned into the default number of volumes (with default sizes), or if it has volumes which have been coalesced and you don't need to split them, then you can proceed to step 6.

4. If the disc has *not* yet been initialized, then you will need to initialize it now.

   Since the disc has not been initialized, the standard TABLE program will not have found *any* valid directories at expected locations on the disc, and therefore will assume that it is to be partitioned into the default number of volumes (shown in the discussion of partitioning given in The Booting Process early in this chapter). It will build the Unit Table accordingly.

   a. Put the ACCESS:MEDIAINIT.CODE program on-line, and press [X] to execute it. The system responds:

      ```
      Execute what file?
      ```

      If the program is on the ACCESS: disc, enter:

      ```
      ACCESS:MEDIAINIT
      ```

      The ".CODE" suffix will automatically be appended to the file name.

      The program prompts for a unit number:

      ```
      Volume ID?
      ```

      Enter the unit number of the first volume on the disc that is to be initialized. In this case, enter:

      ```
      #11:
      ```

      The program asks for verification:

      ```
      Device: 913xA series hard disc, 707, 0
      Logical unit #11 - < no directory >

      WARNING: the initialization will also destroy:
        #12: < no dir >
        #13: < no dir >
        #14: < no dir >

      Are you SURE you want to proceed? (Y/N)
      ```

The 913xA corresponds to the 913x "V" suffix drives. The select code and HP-IB address (707), and drive number (0) should also correspond to the disc to be initialized. If not, then answer [N] and correct the problem. If this is the disc you want to initialize, then answer affirmatively by pressing [Y]. The program then displays:

```
Medium initialization in progress
```

The program takes about 15 minutes to initialize this type of disc.

After the program has finished, it displays this message:

```
Medium initialization completed
```

Each volume's directory is then "zeroed" (cleared, named, and validated):

```
Volume zeroing in progress
```

Here is the message that indicates the entire initialization and zeroing is successful:

```
Volume zeroing completed
```

b. Verify that the volumes are have been zeroed and are accessible by using the Filer's Volumes command. Press [V], and the Filer shows you the volumes currently on-line:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 # V11:
 12 # V12:
 13 # V13:
 14 # V14:
Prefix is - BOOT:
```

b. If all volumes have been initialized and directories zeroed properly, proceed to step 6.

5. If volumes on the disc have been coalesced and you want to split them (or if the disc was initialized as one large volume), then you will need to restore part or all of the default partitioning structure now.

   To do this, you will need to destroy the existing directories on the disc which are to be split. For instance, if your disc is one large logical volume, then you merely need to destroy the first directory, since it is the only directory that currently exists on the disc.

   a. To destroy a directory on the disc, you will overwrite the directory with a file (the MEDIAINIT.CODE file will work for this purpose). Make sure that the Filer is on-line and then invoke it by pressing [F] from the Main Level. Then use the Filecopy command to overwrite the directory with the file; press [F] and the following prompt is displayed:

      ```
      Filecopy what file?
      ```

      Enter:

      ```
      ACCESS:MEDIAINIT.CODE
      ```

It then asks:

```
Filecopy to what?
```

You will answer:

```
#11:
```

The Filer verifies with this prompt:

```
Destroy EVERYTHING on volume V11 ?
```

Affirm that you want to destroy the directory by pressing [Y].

The Filer then shows that it has made the requested copy:

```
ACCESS:MEDIAINIT.CODE        ==> #11:
```

b. If other existing directories on the disc are to be split, then destroy each by repeating this process.

c. After destroying all directories to be split, run TABLE again to restore the Unit Table to set up the default partitioning. This step does **not** partition the disc; it "partitions" the Unit Table in anticipation of the subsequent disc partitioning.

d. Now partition the disc by zeroing volumes #11, #13, and #14. If you are not still in the Filer, put it on-line and press [F]. Use the Zero command to zero the first volume on the disc. Press [Z]. The Filer responds with this prompt:

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

Enter the unit number of the first volume:

```
#11:
```

The Filer then responds:

```
Destroy EVERYTHING on volume V11 ? (Y/N)
```

Press [Y] to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

```
Number of directory entries?
```

If you want a number other than the default (80), then enter it now; otherwise, press [Return] or [ENTER] to accept the default.

The Filer next prompts for the volume size. The number shown in parentheses is the default:

```
Number of bytes (1206272) ?
```

Accept the default by pressing [Return] or [ENTER].

Finally, you will be prompted for the new volume name:

```
New directory name ?
```

Enter any valid volume name of up to 6 characters. For this example, enter:

```
V11:
```

The Filer verifies that it has the name you requested:

```
V11: correct ? (Y/N)
```

Press [Y] to confirm the name, if it is correct. If is not, then answer [N]; you will need to start the Zero command again.

If you confirmed the name, the Filer shows that the directory was created:

```
Volume V11 zeroed
```

e. Repeat this process for each unit number to Zero all directories which you want to *remain* on the disc; you need not Zero those that will be coalesced later in this procedure.

6. If the second directory exists (unit #12), then destroy it. If it does not exist, then proceed to step 7.

   To destroy a directory, use the Filer's Filecopy command to copy a file into the directory (the MEDIAINIT.CODE file will work just fine for this purpose). Make sure the Filer is on-line, and press [F] to enter the Filer. Invoke the Filecopy command by pressing [F]. It prompts:

   ```
   Filecopy what file?
   ```

   Enter the specification of the MEDIAINIT file:

   ```
   ACCESS:MEDIAINIT.CODE
   ```

   The system prompts:

   ```
   Filecopy to what?
   ```

   Enter the specification of the second directory:

   ```
   #12:
   ```

   Since a directory already exists on this volume, the Filer prompts to see if you really want to proceed and destroy this directory:

   ```
   Destroy EVERYTHING on volume V12 ? (Y/N)
   ```

   Type [Y] to enter an affirmative response. The Filer then shows that it completed the operation by displaying this message:

   ```
   ACCESS:MEDIAINIT.CODE              ==> 12:
   ```

7. Now you should execute TABLE again. This execution of the program will find no second directory, and consequently will make the Unit Table entry for the first directory reflect the size of both first and second volumes (about 2 Megabytes). No changes to the disc will be made by this step, however.

8. Now destroy the first directory and then Zero the volume. Destroying this directory is necessary in order to make the Zero command read the size of the volume *from the Unit Table*. If it is *not* destroyed, then the volume size will be read from the disc and the volumes will *not* be coalesced; the first directory will retain its original size.

   a. Use the Filecopy command to overwrite the first directory. While in the Filer, press [F]. The Filer prompts:

      ```
      Filecopy what file?
      ```

      Answer:

      ```
      ACCESS:MEDIAINIT.CODE , #11:
      ```

      The Filer then asks:

      ```
      Destroy directory V11 ? (Y/N)
      ```

      Answer [Y] to affirm that you do want to destroy it.

   b. Finally, Zero unit #11's directory. Press [Z], and the Filer prompts:

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

      Answer:

      ```
      #11:
      ```

      The Filer then responds:

      ```
      Destroy V11 ? (Y/N)
      ```

      Press [Y] to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

      ```
      Number of directory entries?
      ```

      If you want a number other than the default (80), then enter it now; otherwise, press [Return] or [ENTER] to accept the default.

      The Filer next prompts for the volume size. The number shown in parentheses is the default (note that it is now twice its former size):

      ```
      Number of bytes (2412544) ?
      ```

      Accept the default by pressing [Return] or [ENTER].

      Finally, you will be prompted for the new volume name:

      ```
      New directory name ?
      ```

      Enter any valid volume name of up to 6 characters. For this example, enter:

      ```
      V11:
      ```

      The Filer verifies that it has the name you requested:

      ```
      V11: correct ? (Y/N)
      ```

      Press [Y] to confirm the name, if it is correct. If is not, then answer [N]; you will need to start the Zero command sequence again.

If you confirmed the name, the Filer shows that the directory was zeroed:

```
Volume V11 zeroed
```

9. After zeroing has completed, verify that the disc is partitioned as desired.

   a. Use the Filer's Volumes command to verify that there are only volumes #11, #13:, and #14: on the disc.

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 # V11:
 13 # V13:
 14 # V14:
Prefix is - BOOT:
```

   If you don't have the partitioning scheme that you want, you may have made a mistake during the procedure. You will need to repeat the appropriate part(s) of the procedure.

   b. Use the Filer's List_directory command to verify that volume #11: is now larger. Look for the number of available sectors (it should be approximately as shown below).

```
V11:                Directory type= LIF level 1
created 15-Jan-87   1.31.24 block size=256
 Storage order
...file name....     # blks     # bytes  last chng

FILES shown=0 allocated=0 unallocated=80
BLOCKS (256 bytes) used=0 unused=9412 largest space=9412
```

   Note that the number of entries you specified for the directory will affect the number of sectors usable for files. This example shows the number of sectors left after allocating space for 80 directory entries (files). You will have fewer sectors usable for files if you specified a greater number of entries.

   If the size is not what you expected, then you may have made a mistake during the procedure. If so, you will need to repeat the appropriate part(s) of the procedure.

Copying System Files and Changing Their Names

One of the easiest ways to change the configuration of your system is to copy files from the flexible discs on which it is shipped to mass storage with better performance (such as local hard discs or SRM). This section describes several things to consider while making this type of modification to your system.

Copying Files to the System Volume

If you have a system with one hard disc (such as a CS80 or 913x hard disc), a double-sided flexible disc (such as the 9122), or an eight-inch flexible disc (such as the 9895), then that device may be selected as the system volume during the boot process. (The system volume and its uses are described in the *Pascal User's Guide* and in the File System and Filer chapters of Volume I of this manual.) It is often useful to copy the most-used of the following "system files" to this system volume to increase performance.

```
EDITOR
FILER
COMPILER
LIBRARY
LIBRARIAN
ASSEMBLER
```

If you P-load these files, then you may not want them to take up room on your system volume; you may want to put them on another less-used volume.

As mentioned at the beginning of this chapter, the following files are used during the boot process. They are on the standard BOOT: disc shipped with your system.

```
SYSTEM_P
INITLIB
TABLE
```

If you have Boot ROM 3.0 or later versions (and not 3.0L), these BOOT: files may be placed together on any hard volume you choose, provided it is LIF or HFS formatted, or is an SRM. They can also be renamed; the purposes of and conventions for renaming these BOOT: files will be described later in this section. If you do not have Boot ROM 3.0 or later version (which is only possible with earlier 9826 and 9836 computers), these files must all be on the right-hand internal disc drive. (The method of determining which Boot ROM you have is described in the *Pascal User's Guide.*)

The STARTUP file is also used during the boot process, but it can either be on the boot volume or on the system volume. You might leave it on the boot volume if there isn't room on the system volume. However, if possible, put it on the system volume so that it loads faster.

If you have an HP 9885 8-inch disc drive as the system device, not all the system files will fit on it. Using the above procedure, copy onto it those system files which are used most frequently (such as EDITOR, FILER, and COMPILER).

Default BOOT: File Names

On the BOOT: discs shipped from the factory, the files used during the boot process are named as follows:

```
SYSTEM_P
TABLE
INITLIB
STARTUP
```

If these files are copied to another WS1.0 or LIF mass storage volume and the names are retained, they will boot normally. HFS discs will not boot until the SYSTEM_P file is installed in the boot area of the disc by the OSINSTALL utility. See Chapter 21 of this manual.

Re-Naming the BOOT: Files

If you change the name of SYSTEM_P (the system Boot file), then you must also change the names of some other files on the BOOT: disc. The advantage is that the same Boot file (with different names) can load specialized BOOT: files for unique hardware configurations.

Note

The term "BOOT: file" is used to identify a file used during the boot process; these files are on the standard BOOT: or BOOT2: disc shipped with your system.

The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.

The Boot ROM 3.0 and later versions also recognize system Boot files if the file name begins with "SYS". If you rename the Pascal system Boot file (SYSTEM_P), there are file naming rules you must follow so the system Boot file can identify the other BOOT: files. The rules for non-standard BOOT: file names are as follows:

- If the complete string "SYSTEM_" is used in the system Boot file name, up to the next three letters of the file name are added to the base of the other BOOT: file names (INIT, TABLE, START).
- If only "SYS" is used in the system Boot file name, up to the next seven letters of the file name are added to the base of the other boot file names (possible only with Boot ROM 3.0 and later versions).

For example, if you change the system Boot file's name from SYSTEM_P to SYSTEM_P3 (for Pascal 3.0), then the Boot file will look for the following files:

```
INITP3
TABLEP3
STARTP3
```

Keep in mind that file names on a LIF directory must be 10 characters or less, and 14 characters or less on an HFS directory.

If you change the name to Boot file's name SYS_SRM_P3, it will look for the following files:

```
INIT_SRM_P3
TABLE_SRM_P3
START_SRM_P3
```

File names on SRM directories can be up to 16 characters.

In general, SYSTEM_P could be the Pascal Boot file that loads the standard Pascal BOOT: files. SYSTEM_P3 is the same Boot file, but INITP3 and TABLEP3 could support the hard discs. SYS_SRM_P3 is the same Boot file, but INIT_SRM_P3 and TABLE_SRM_P3 could support SRM.

Normally, a special TABLE is not required for hard discs or SRM systems, although you may wish to create one for a special application.

SRM users needing a special TABLE should use the "private" system volume based on the node address mechanism to keep their own custom TABLE and INITLIB (i.e., the default system volume when booting a workstation from SRM is the /WORKSTATIONS/SYSTEM*nn* directory, in which *nn* is the node address of the workstation). See the relevant SRM documentation for further details.

If you have an HFS disc, which like SRM uses a hierarchical file system, it is possible to use a similar method to that described above for SRM. However, with HFS there are no node addresses; when booting a workstation from an HFS disc you have two choices:

1. Leave SYSTEM_P unchanged. The computer will look in `/WORKSTATIONS/SYSTEM` for INITLIB, TABLE, STARTUP, and may look again in the * volume for STARTUP. If these files are not found, the boot may not be successful.

2. Change the name from SYSTEM_P to SYSTEM_xxx or SYSxxxxxxx. Representing either suffix by xxx, the system will first look in `/WORKSTATIONS/SYSTEMxxx` on the boot disc for INITxxx, TABLExxx, and STARTxxx. If these files are not found, it will look in `/WORKSTATIONS/SYSTEM` for the same file names.

   For example, if the boot file is named: `SYSP32FHS`, the system will try for: `INITP32HFS`, `TABLEP32HFS`, and `STARTP32HFS` in the directory `/WORKSTATIONS/SYSTEMP32HFS`. If they are not found, the system will try for the same names in the `/WORKSTATIONS/SYSTEM` directory.

AUTOSTART and AUTOKEYS Stream Files

Stream files allow execution of commands just as if you had entered them from the keyboard. When you put a Stream file named AUTOSTART on your system volume, the keyboard commands the file contains are automatically executed during the booting process; if the volume is a read-only device, such as EPROM, then you should call the stream file AUTOKEYS. (The Stream command is fully described in the Main Command Level chapter of Volume I of this manual.)

You can use autostart files to perform such functions as the following: load drivers; use the What command to change the system files, system library, default or system volume; P-load programs, re-execute the TABLE program (or execute another like it). Be aware that there are also other ways to perform this type of configuration; however, this method can be used to quickly or temporarily change the configuration by creating different stream files, renaming the one you want to be used as the autostart file to `AUTOSTART` (or `AUTOKEYS`), and then re-booting.

Autostart files are also useful for setting the timezone in version 3.2.

For example, in order to configure your system to access an HP 98626 RS-232C Serial interface and set the timezone, you can use an autostart file. Here is an example stream file that does this (this example assumes that the file named RS232 is on the volume that is chosen as the system volume at power-up):

```
<blank line>
[7]
x*RS232.
v
```

The blank line which occurs first is a carriage return in the file which is a "null" response to the Date prompt at power-up. The next line sets the Time prompt's Timezone to MST (Mountain Standard Time) which is +7 hours from GMT (Greenwich Mean Time). These two responses get you to the Main Command Level. For details on these two lines, see the `VERSION` command in the *Pascal 3.2 Workstation System Volume 1*, Main Command Level chapter.

The "x" in the first column of the third line is an eXecute command. The period at the end of the file name prevents the system from appending ".CODE" to the file name. The "v" at the end of the file is the Version command. It gives you another chance to type in the time and date.

After you've created your AUTOSTART file, be sure that you store it on the system volume. This is done by Quitting the Editor, selecting the Write option, and entering this file specification:

```
*AUTOSTART.
```

The period at the end of the file name prevents ".TEXT" from being added to the file name. If you were a Pascal 1.0 user, the file was called AUTOSTART.TEXT. With Pascal 2.0 and later versions, it is called AUTOSTART. Notice that uppercase characters must be used.

Adding Modules to INITLIB

As mentioned previously, the INITLIB supplied on your original BOOT: or BOOT2: disc contains a reasonably complete set of peripheral driver software. You may wish to install other drivers, which are supplied on the CONFIG: or LIB: disc (ACCESS: for double-sided media); or to conserve memory you may wish to remove items you don't need.

Unlike the System Library, modules in INITLIB are order sensitive. Certain modules, if present, must precede others in INITLIB. The list which follows shows the recommended order of all the "driver" modules supplied with Pascal 3.2. If you add or delete INITLIB modules, all the modules which are present in the resulting INITLIB should appear in the order listed.

Required Order of Modules in INITLIB

The table lists the importance of each module. Items marked "Required" are essentially required in INITLIB. Items marked "Almost" are almost always required. These modules should not be removed unless you have determined for sure they aren't needed, because they are part of the normal functioning of the system. Items marked "Development" are usually needed in a software development environment. Items marked "Optional" are optional unless required by a particular system configuration; the hardware or application which requires them is usually noted in parentheses in the "Importance" column. Also, CRT drivers are briefly described.

Required Order of Modules

Module	Where found	Importance
KERNEL	BOOT:INITLIB	Required
SYSDEVS	BOOT:INITLIB	Required
CRT	BOOT:INITLIB	Required for the 98546A display and all Series 200 except Model 237
CRTB	BOOT:INITLIB	Required for Model 237 display
CRTC	BOOT2:INITLIB	Required for Series 300 except 98700 and 98546A displays
CRTD	BOOT2:INITLIB	Required for 98700 display
CRTE	BOOT2:INITLIB	Required for the following Series 300 displays: 98548, 98549, and 98550.
A804XDVR	BOOT:INITLIB	Required
KEYS	BOOT:INITLIB	Required
NONUSKBD1	BOOT:INITLIB	Required for non-US language keyboard
NONUSKBD2	BOOT:INITLIB	Required for non-US language keyboard
BAT	BOOT:INITLIB	Optional (needed for Series 200 only)
CLOCK	BOOT:INITLIB	Required
PRINTER	BOOT:INITLIB	Optional
DISCHPIB	BOOT:INITLIB	Almost (needed for most external discs)
AMIGO	BOOT:INITLIB	Optional (needed for "older" discs)
CS80	BOOT:INITLIB	Optional (needed for "newer" CS-80 and SS-80 tapes and discs)
IODECLARATIONS	BOOT:INITLIB	Required
HPIB	BOOT:INITLIB	Almost (needed for any HP-IB use, including external HP-IB discs and printers)
DMA	BOOT:INITLIB	Optional (needed for 98620 and built-in "DMA-C0" on Models 330 and 350)
REALS	BOOT:INITLIB	Required
ASC_AM	BOOT:INITLIB	Optional (needed for ASCII type files)
WS1.0_DAM	CONFIG:WS1.0_DAM	Optional (needed for Pascal 1.0 discs)
TEXT_AM	BOOT:INITLIB	Almost (needed for TEXT type files)

Module	Where found	Importance
CONVERT_TEXT	BOOT:INITLIB	Almost (needed for EDITOR, COMPILER, and ASSEMBLER)
LIF_DAM	BOOT:INITLIB	Almost
CHOOK	BOOT:INITLIB	Optional (needed for Model 236C)
DEBUGGER	ASM:DEBUGGER	Development
REVASM	ASM:DEBUGGER	Development
DISC_INTF	CONFIG:DISC_INTF	Optional (needed for 98625 interface and built-in hi-speed disc interface on Models 330 and 350)
SCSIDVR	LIB:SCSIDVR	Optional (needed for HP 98658A or HP 98265A SCSI interface)
SCSIDISC	LIB:SCSIDISC	Optional (needed for SCSI discs)
DATA_COMM	CONFIG:DATA_COMM	Optional (needed for 98628 and SRM-98629 interface)
GPIO	CONFIG:GPIO	Optional (needed for 98622 interface)
RS232	CONFIG:RS232	Optional (needed for 98626, 98644, built-in RS-232)
PARALLEL	LIB:PARALLEL	Optional (needed for built-in HP parallel interface)
LAN	ACCESS:LAN	Optional (needed for 98643 and built-in LAN)
VMELIBRARY	ACCESS:VMELIBRARY	Optional (needed for 98646A)
SRM	CONFIG:SRM	Optional (needed for 98629 interface)
F9885	CONFIG:F9885	Optional (needed for 9885 disc)
BUBBLE	LIB:BUBBLE	Optional (bubble mass storage)
EPROMS	LIB:EPROMS	Optional (EPROM mass storage)
EDRIVER	LIB:EDRIVER	Optional (EPROM programming)
SEGMENTER	CONFIG:SEGMENTER	Optional
HPHIL	CONFIG:HPHIL	Optional (needed for non-keyboard HP-HIL devices)
MOUSE	CONFIG:MOUSE	Optional (arrow key generation)
DGL_ABS	CONFIG:DGL_ABS	Optional (HP-HIL tablets for DGL)
DGL_REL	CONFIG:DGL_REL	Optional (HP-HIL mice and external knobs for DGL)
HFS_DAM	HFS:HFS_DAM	Optional (needed for HFS discs)
UXTEXT_AM	HFS:UXTEXT_AM	Optional (tab char. handling in UX type files)
LAST	BOOT:LAST	Required

Note: The BOOT2:INITLIB file contains most of the modules in the BOOT:INITLIB files (the only exceptions are the CRT, CRTB, CHOOK, and BAT modules).

Individual Module Descriptions

Here are brief descriptions of each of the above modules.

- KERNEL is the "core" of the system, containing the Library facility for the Linking Loader and basic File System support. It is always required.
- SYSDEVS, CRT, CRTB, CRTC, CRTD, CRTE, A804XDVR, KEYS, NONUSKBD1, NONUSKBD2, BAT, and CLOCK are responsible for the CRT display, keyboard, foreign character set, Series 200 battery backup, and clock. They are broken out into several small modules so they may be replaced individually if desired.

  Here are descriptions of the CRT modules:

  CRT Used only with Series 200 displays with *separate* alpha and graphics planes (that is, with all Series 200 computers except the Model 237). Used also with the Series 300 HP98546 Display Compatibility Interface.

  CRTB Used only with the Model 237 display.

  CRTC Used in the following Series 300 displays: HP98542, 98543, 98544, 98545, and 98547.

  CRTD Used only with the HP98700 Display Controller.

  CRTE Used with the following Series 300 displays: HP98548, 98549, and 98550.

  Modules NONUSKBD1 and NONUSKBD2 need only be present or replaced by code with equivalent function if non-US-ASCII keyboards are used. To be a little more specific, NONUSKBD1 is required for 98203A,B,C non-US keyboards and Katakana keyboards, and NONUSKBD2 is needed for non-US 46020 and 46021 keyboards *except* Katakana.

  Module CLOCK provides date and time computations and initializes the time for battery-backed clocks.

  Note

  You can also IMPORT the SYSDEVS module (i.e., use data objects and code declared in it), which is described in the Procedure Library manual. This is the only system module that is described fully enough in this manual set to use in this fashion.

- PRINTER is required to drive all printers, regardless of the type of interface electronics being used. It supports serial, parallel, and HP-IB printers; however, you will have to use the RS232 driver module in order to use printers with RS-232C interfaces or the PARALLEL module for printers with parallel interfaces. You may also have to modify the variable named `local_printer_timeout` in the CTABLE program; see the discussion of modifying CTABLE later in this chapter.

- DISCHPIB, AMIGO, and CS80 modules are related. To use any external disc drive connected via HP-IB you must use the following modules:

  DISCHPIB and AMIGO for these disc models:

  9895 8-inch flexible disc

  9121 single-sided 3.5-inch flexible disc

  913x small hard discs

  8290x 5.25-inch flexible disc

  Note

  HFS is not supported on the HP 9885 8-inch flexible disc drive, nor on removable media drives that are accessed by the AMIGO driver module. This includes the HP 9895 8-inch drive, the HP 82901 and HP 82902 5.25-inch drives, and the HP 9121 3.5-inch drive. Also not supported by HFS is the removable media unit in AMIGO "multiple-unit" drives such as the HP 9135 and the HP 9133A, B, C, and XV. However, the hard disc unit in such a multiple-unit drive **can** be used as an HFS unit.

  DISCHPIB and CS80 for fixed discs of the Command-Set/80 (CS80) and Sub-Set/80 (SS80) disc drives, and cartridge tape drives, including these models:

  79xx large hard discs; optional integrated cartridge tapes

  9122 double-sided 3.5-inch flexible disc

  913x not-so-small hard discs

  9144 stand-alone DC600 tape drive

  9153x hard discs

  Module HPIB is also required for all the above disc/tape drives when used on built-in HPIB interfaces or on 98624 HP-IB interfaces.

- IODECLARATIONS is the lowest level of device IO support. Although it is possible to construct loadable systems without this module, only the internal disc drives on the Model 226 and Model 236 can be accessed.

- HPIB is the lowest level support for the Hewlett-Packard Interface Bus, which is HP's implementation of the IEEE-488 Standard. HP-IB interfaces include the built-in HP-IB and HP 98624 cards. Most HP peripherals have HP-IB interfaces, so you will rarely remove this module.

- DMA is the module which runs the HP 98620 Direct Memory Access interface card and the built-in "DMA-C0" on the Models 330 and 350. DMA provides very high speed data transfers. It is also required in order to use the HP 98625 Disc Interface or the built-in DMA hardware of some Series 300 computers.

- REALS is the floating-point mathematics support package. It also supports the HP 98635A Floating-Point Math card.
- ASC_AM is the Access Method responsible for blocking and unblocking text files with the LIF-ASCII structure (.ASC files). LIF stands for Logical Interchange Format, a common file interchange structure supported by many HP computers. Since this is one of the formats used by the BASIC language system, it is a good thing to have around. It is also the format used by the SRM for spooled printer files.
- WS1.0_DAM is the Directory Access Method used by the Pascal 1.0 system, a predecessor to the one you are using. This module lets the system read and write discs in that format. Note that the WS1.0 disc organization is compatible with discs written by UCSD Pascal systems; but to read discs written by non-HP computers, a special disc driver is usually required. This DAM can be left out if you have no need to read or write discs compatible with the Pascal 1.0 system.
- TEXT_AM is the Access Method used to block and unblock text files created with the ".TEXT" suffix. These are the files normally created by the Editor (unless the user specifies otherwise). The ".TEXT" file structure is compatible with text files generated by UCSD Pascal systems.
- UXTEXT_AM is an augmented Access Method for HP-UX compatible text files (suffix .UX). Although handling of UX files is built into Pascal 3.2, tab character expansion is not. With UXTEXT_AM, tabs are automatically expanded to 8-position tab stops upon input from UX type files. UXTEXT_AM has no effect on item-oriented files. Note that UX files can also reside on WS1.0, LIF, or SRM mass storage.
- CONVERT_TEXT is a module used by the Compiler and other subsystems to convert among the various representations of text files. It should be present in INITLIB. It will expand tabs on input from UX files, for convenience.
- LIF_DAM is the Directory Access Method required to read and write HP Logical Interchange Format disc directories. LIF is the primary directory organization used with Pascal 2.0 and later system versions, so this module is normally present. If you configure your system to use WS1.0 or HFS as the primary directory method (as described in the Special Configurations chapter), you may remove LIF_DAM.
- CHOOK is the dump graphics driver for the Model 236C. It also supports alpha/graphics toggling on the Model 236C CRT.
- DEBUGGER is the interactive debugging tool. It is not part of INITLIB (as in pre-3.0 system versions) due to lack of disc space and because it is a particularly dangerous thing to put in the hands of non-programmers. Module REVASM is also a handy tool to have while debugging programs; it allows you to display the contents of memory locations as Assembler language instructions (i.e., "reverse assemble" them).
- REVASM — see DEBUGGER above. Only useful with DEBUGGER.
- DISC_INTF and DMA modules are required in order to use the HP 98625 High-Speed Disc interface (or the built-in high-speed disc interface of the Models 330 and 350) in conjunction with the DMA hardware. The 98625 interface requires DMA hardware.
- SCSIDVR is the module required in order to use the HP 98658A, HP 98265A, or the built-in SCSI interface. The DMA module and DMA hardware will enhance performance.

- SCSIDISC is the module required in order for the system to access SCSI discs.
- DATA_COMM is the module required to drive HP 98628 Data Comm and HP 98629 SRM interfaces.
- GPIO is the module required to drive the HP 98622 GPIO (16-bit parallel) interface.
- RS232 is the module required to drive the HP 98626 and 98644 RS-232C Serial interfaces, and the built-in serial interface in any of the Series 200/300 computers.
- PARALLEL is the module required to drive the built-in HP parallel interface.
- LAN is required to drive the HP 98643 interface and built-in LAN interfaces.
- SRM is required to drive the HP 98629 SRM interface. Module DATA_COMM is also required when using SRM.
- F9885 is required for model 9885 flexible disc drives. These discs also require the DMA module, HP 98620 DMA card, and HP 98622 GPIO interface.

Note

HFS is not supported on the HP 9885 8-inch flexible disc drive.

- BUBBLE is the module that drives HP 98259 Magnetic Bubble Memory cards. In order to use these cards for mass storage, you will need to add this module to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- EPROMS is required to access the HP 98255 EPROM cards with the file system. In order to use these cards for mass storage, you will need to add this module to INITLIB, modify the TABLE program, and program the EPROMs using the HP 98253 EPROM Programmer card (which requires the EDRIVER module and the ETU utility). See the Using Bubbles and EPROMs section for further details.
- EDRIVER is a library required to program EPROMs using the HP 98253 EPROM Programmer card. It is used by the ETU utility. In order to use EPROM cards for mass storage, you will need to add module EPROMS to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- SEGMENTER provides the ability to segment programs and run each segment separately. See the Segmentation Procedures chapter of the *Pascal Procedure Library* manual for further details.
- HPHIL provides the drivers for HP Human Interface Link devices (except keyboards); it is an extension to the A804XDVR module. You can remove it if your computer does not have one of these devices (for example, a mouse input device).
- MOUSE provides a driver for the optional "mouse" input device, the HP-HIL knob, and the knob in the 98203C keyboard, all of which can be connected to the computer through the HP Human Interface Link (HP-HIL). The driver supports using the device for "arrow-key" cursor-movement input in both horizontal and vertical directions. The MOUSE module requires the HPHIL module.

- DGL_ABS provides DGL support for the 46087, 46088, and 45911A graphics tablets and 35723 TouchScreen input devices. The DGL_ABS module requires the HPHIL module.
- DGL_REL provides optimized DGL support for the mouse, HP-HIL knob, and knob on the 98203C keyboard. The DGL_REL module requires the HPHIL module.
- VMELIBRARY contains the modules which drive the VME 98646A card. This file will have to be added to INITLIB or `P-loaded` before using an application which uses this card. See the *Pascal 3.2 Procedure Library*, VME chapter for further details.
- HFS_DAM is required to be able to read and write HP Hierarchical File System format disc directories. As of version 3.2 the HFS DAM is the standard secondary DAM.
- LAST is required in every case, and *must* be the last module in INITLIB. The purpose of this module is to actually start the system running after the contents of INITLIB have been loaded and installed in memory. LAST principally does two things: it loads and executes the configuration file called TABLE; and it loads and executes a file called STARTUP, which is usually the Command Interpreter but may be a user program.

A Note About the INTERFACE File

INTERFACE contains only the interface text of operating system modules in INITLIB (the code was loaded at boot time). You will need to make this interface text available to the Compiler when you import any of the modules in INITLIB; you can do this by copying the module to be imported from CONFIG:INTERFACE (or ACCESS:INTERFACE) to the System Library or using a SEARCH Compiler option that specifies the INTERFACE file. A good example is importing the SYSGLOBALS module, which requires that INTERFACE be accessed by the Compiler. See the *Pascal Procedure Library* manual for further details.

Note INTERFACE also contains the interface text of the SYSDEVS module. Using procedures, etc. from this module is described in the *Pascal Procedure Library* manual. Access to most other modules in INTERFACE is not described in the current Pascal documentation set.

Steps for Adding Modules to INITLIB

For this example, we will add module RS232 (on the CONFIG: disc or the ACCESS: disc) to the INITLIB file that you are now using to boot your system (possibly on the BOOT: disc); actually, you will create a new INITLIB that includes all existing drivers plus this additional driver module. This is the module required to access the HP 98626, HP 98644, or (on some models) the built in RS-232C Serial interface cards.

Here is an outline of the procedure that you can use for adding driver modules to the INITLIB library file. It is a straight-forward usage of the Librarian.

1. Set up mass storage. You will need enough on-line mass storage to store *two* copies of the INITLIB file: one for the source (existing) copy, and one for the destination (new) copy. This requirement is made because the new copy of the INITLIB file must not be taken off-line during the whole process.

   To satisfy this requirement, you will minimally need one of the following configurations: one disc large enough to store both (such as a hard disc or double-sided flexible disc); two flexible disc drives; a flexible disc drive and a memory volume. If you have a hard disc, space is usually not a problem.

2. Copy all modules except LAST from the source INITLIB file into the new INITLIB file on the destination disc. (You may also remove modules from it as it is copied, if desired.)

3. Add the RS232 module to the INITLIB file on the destination disc.

4. Copy the module named LAST from the source INITLIB to the destination file.

5. Replace the existing INITLIB with the new file.

Mass Storage Requirements

As shipped, the INITLIB file requires about 760 sectors (195 Kbytes) of disc space. Since you will have two copies of this file on-line, and one cannot be removed during the process of adding modules to it, here are the mass storage requirements:

- If you are using small-capacity flexible disc drives (approximately 270 Kbytes per disc), then you will need either two drives or one drive and a memory volume.

- If you are using an SRM shared disc, or have a local hard disc (all have volumes with capacities of 1 Mbyte or greater) or a double-sided flexible disc (approximately 630 Kbytes per disc), then you will need only one disc.

You may also need to initialize one or two blank discs (using the MEDIAINIT.CODE utility) then label them. Disc initialization is discussed in the *Pascal User's Guide.* Write the name on a label before applying the label to the disc; sharp instruments are likely to damage the disc. You will also want to make a back-up copy of the BOOT: disc on which the INITLIB file resides.

Making a Memory Volume

If you only have one small-capacity flexible disc drive, then you will need to make sure that you have enough memory to make a memory volume of sufficient size. Here is how to determine the amount of memory in your computer. If the machine is on, turn it off, along with any disc drives to which it is connected. Open the doors of any built-in flexible disc drives. If the SRM is already connected, remove the cable connected to the System Resource Management interface in the back. Now turn the computer on. After going through self-test, the CRT will display the amount of available memory. If you have at least 524 000 bytes, there is "enough" memory to proceed with only one small-capacity disc drive. After turning off computer power, you can reconnect cables and then turn on any peripherals connected to your computer and reboot.

If you determined that you have "enough" memory and must use some memory for mass storage, the following steps are necessary.

1. At the Main Command Level, press [M]. The computer responds:

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

   What unit number?
   ```

2. Enter: #50 [Return]

   The computer then asks:

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

3. Enter: 520 [Return]

   The computer asks:

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

4. You answer: 8 [Return]

   The computer finishes:

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

5. Use the Filer's Change command to re-name the memory volume from RAM: to DEST: (which is the volume name assumed in the following procedure).

This has reserved 266 240 (520*512) bytes of memory to use as a mass storage device. It is like having a 5.25-inch disc drive with a disc named DEST: inserted in it.

Note

If you want to make an HFS memory volume, you need to perform two further steps. These are running the MKHFS Utility and then running the TABLE program in order that the System recognizes the HFS format of the memory volume. However, an HFS memory volume has more overhead, and typically demands more blocks of RAM to provide the same user storage capacity as LIF.

Assumptions Made During this Procedure

Now you are ready to start the process of making the new INITLIB file. This procedure makes the following assumptions:

- The existing INITLIB file is on the BOOT: disc.
- The destination volume is called DEST:.
- The RS232 driver module is on the CONFIG: disc.

1. Make sure that the Librarian is on-line (or insert the ACCESS: disc into the drive you have been using) and press [L] to load it.
2. When you see the Librarian's prompt line at the top of the CRT, press [O] to specify the name of the (Output) file the Librarian will be creating. Enter this name as the destination (new library's) file name:

   ```
   DEST:INITNEW
   ```

   Note

   If you are using a flexible drive, you must not remove the Output disc until the end of this procedure, after you have Quit the Librarian.

3. Press [I] so you can specify an Input file, then enter:

   ```
   BOOT:INITLIB.
   ```

   If the file is not on the BOOT: volume, then you will need to change the leading volume specification. Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the Input file.

4. Near the bottom of the screen you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press [T] to transfer this module to the output file. After a few moments, the name of the next module will appear (probably SYSDEVS). Each time a new module name appears, press [T] to transfer it to the Output file. You should continue copying modules until the name LAST appears. **Don't copy module LAST yet.**

5. Now you must get the required RS232 drivers from the CONFIG: disc (or ACCESS: disc for double-sided media) and transfer them to the Output file. If the CONFIG: disc (or a disc containing the RS232 module) is not currently on-line, then put it on-line. Press [I] for an Input file and enter this file specification:

   ```
   CONFIG:RS232.
   ```

   (If you are not copying the module from the CONFIG: disc, then use the volume specification of your source disc.) Don't forget the period after the file name.

6. When the module name RS232 shows up near the bottom of the screen, press [A] which tells the Librarian to transfer All the modules in the file. Remove the CONFIG: disc after the module has been transferred.

7. Put in BOOT: once more, press [I] for Input, and enter the file specification of the original INITLIB file:

   ```
   BOOT:INITLIB.
   ```

8. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing [M] for module and enter:

   ```
   LAST
   ```

   Then transfer LAST to the Output file by typing [T].

9. You now have all the modules in your new library. "Keep" it by typing [K].

10. You may want to verify that these modules are in the new library. Press [I] and specify the new library as the Input file:

    ```
    DEST:INITNEW
    ```

    (The .CODE will automatically be appended to the file name.) Step through the file with the space bar. If all modules are present, then Quit the Librarian by typing [Q], else redo the procedure.

11. Remove the old copy of INITLIB from the BOOT: disc with the Filer's Remove command (if the Filer is not on-line, you will need to put it on-line before trying to invoke it). Then Krunch the BOOT: disc so that you will have enough room on the disc to store the new, larger copy of the INITLIB file (INITNEW.CODE).

12. Use the Filer's Filecopy command to copy the new library file (DEST:INITNEW.CODE) onto the BOOT: disc, changing the name INITNEW.CODE to INITLIB as you copy it.

13. The next time that you boot your system with this new INITLIB, module RS232 will automatically be installed.

Modifying the TABLE Program

This section first describes the structure of the TABLE program. If you want to change something that it does, you will need to edit and re-compile the CTABLE.TEXT source program on the CONFIG: disc. For double-sided media this program source can be found on the ACCESS: disc.

Overview of General Steps

Here are the general steps you should take to modify TABLE:

1. The Pascal source of TABLE, called CTABLE.TEXT, is provided on the CONFIG: disc distributed with every copy of the system. Read the commentary on the CTABLE source program in this section. You should follow along in the source program as you read the corresponding commentary.
2. Make your modifications to a *copy* of the program — **not the original.**
3. Compile this modified program, which yields an object code file (for instance, MYTABLE.CODE).
3. Execute the modified program to see if the results are correct. In fact, the Unit Table can be reconfigured any time by executing a version of TABLE.
4. When you are quite sure the new TABLE program is correct, use the Filer to copy the compiled code to your BOOT: disc. The name of the copy must be TABLE (not TABLE.CODE) in order to be recognized during boot-up (with Boot ROM 3.0 and later, it may begin with the letters TABLE and end with letters that match the other BOOT: file names; see the discussion of Renaming the BOOT: Files earlier in this section).

CAUTION

Be careful here! If there is no backup copy of the BOOT: disc containing the original table, and there is something wrong with the modified table, you may not even be able to use your system.

---

5. Depending on the size of INITLIB, there may not be much room on the BOOT: disc. You may need to Krunch the disc with the Filer to make space. The modified TABLE can also be made considerably smaller by linking it to itself. This combines all the internal modules into a single module, and gets rid of module interface text and internal reference information. The procedure for linking the modules of a file is presented later in this section.

Commentary on the CTABLE Program

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

```
program ctable; {a pseudo-pascal functional description}

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

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

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

    module scanstuff;       {DON'T MODIFY THIS MODULE.}
        {Contains code which asks each HP-IB device to identify itself.}

    module SCSIscanstuff;  {DON'T MODIFY THIS MODULE.}
        {Contains code which asks each SCSI device to identify itself.}

begin
    initialize hardware (interfaces, etc.).
    assign default 'device address vectors'.
    scan for devices on various HPIB addresses.
    scan for devices on various SCSI addresses.
    scan for an internal mini-floppy drive.
    determine the nature of the boot device.
    create temporary unit table.
    make 'standard' assignments #1:-#6:
      (in temporary Unit Table).
    assign units #7:-#10: (to 2nd and 3rd priority floppies).
    assign units #11:-#40: (to local hard discs).
    assign units #41:,#42: (to tape drives).
    assign units #43:-#44: (as alternate DAMs for #3:-#4: floppies).
    assign unit #45: as additional entry for SRM
       (note the template for #46).
    assign units #47:-#50: (as alternate DAMs for #7:-#10: floppies).
    make optional templates for 'manually' overriding preceding defaults
      [hard-disc partitioning, tape drives, EPROM and Bubbles, etc.].
    copy temporary Unit Table to actual system Unit Table.
    prefix directory on SRM for #5: (default) and #45: (system).
    remove extraneous local hard disc entries if necessary.
    assign unit for system volume.
    set prefix of SRM default volume.
    set up unit #46 as sysvol if booted from HFS fixed disc.
    make optional templates for extra HFS unit entries.
    re-open the 'standard' system files (#1:, #2:, and #6:).
 end.
```

Note

You may want to read through the following discussion of the standard TABLE in its entirety before editing the CTABLE.TEXT source file. On the other hand, you may want to edit it as you read the discussion.

The general recommendation is that you should edit the main program **only** if the desired result cannot be obtained by modifying the declarations in module `options`.

Modifying Module OPTIONS

This module consists only of declarations of exported types and constants. The constants are used by the main program; each section describes their effects and how to modify them.

Secondary Directory Access Method (DAM)

The following constant selects the secondary Directory Access Method for *local* (i.e., non-SRM) mass storage devices.

```
{two possible versions of TABLE}
  const
    hfsversion  = {LIF primary DAM, HFS secondary}
    ucsdversion = {LIF primary DAM, UCSD secondary}
{change this assignment to get different versions}
  const
    thisversion = hfsversion;
```

HP's Logical Interchange Format directory is the primary DAM for either version selected. `hfsversion` specifies the Workstation Hierarchical File System as the secondary DAM. The constant `ucsdversion` specifies the format used in Pascal 1.0 (WS1.0 DAM) as the secondary DAM. Note that this is really only applicable to floppy-disc drives. Depending on the selection of `thisversion`, accessing a floppy disc in a drive using alternate DAM units, e.g. #43, will cause the system to expect either an HFS or WS1.0 file system. The standard TABLE expects that remote mass storage devices will use the Shared Resource Manager's hierarchical (structured) directory format (SDF), so no declaration is needed here.

Power-Up System Volume

The following constant selects the system volume at power-up.

```
{power-up system unit}
  const
    specified_system_unit =
      0; { <>0 overrides auto-assignment}
```

When `specified_system_unit` is zero (the default), the program makes its own choice according to the algorithm described in the preceding discussion of The Booting Process.

If you change this constant to a non-zero value, then it indicates which of the 50 units is to be the system volume. For instance, if you change this constant to 3, then drive #3: will become the system volume. This explicit choice overrides any units specified in the subsequent `system unit auto-search declarations` section of this module.

Floppy Disc Unit Pairs

The standard TABLE program is set up to assign unit numbers to up to three dual floppy disc drives. They will normally occupy units in pairs (#3: and #4:, #7: and #8:, and #9: and #10:). Hard discs usually begin at unit #11: and can be assigned up to 30 unit numbers (up to unit #40:).

```
{floppy/harddisc unit number slot tradeoff's}
  const
    floppy_unit_pairs =  {[1..10]}
      3;
    harddisc_first_lun = {do not edit!}
      7+(floppy_unit_pairs-1)*2;
    harddisc_last_lun =
      40;
```

In order for the TABLE program to assign unit numbers to other than three floppy-disc pairs, you will need to change the `floppy_unit_pairs` variable accordingly. Note that changes to this constant affect the beginning unit assigned to the highest priority hard disc in the system. For instance, changing the constant to 2 causes hard discs to begin at #9, while changing it to 4 causes hard discs to begin at unit #13.

Local Printer Type Option

This constant determines the type of the local printer; it can be either HPIB, RS232, or PARALLEL.

```
{local printer type option}
  type
    local_printer_type = (HPIB, RS232, PARALLEL);
  const
    local_printer_option = HPIB;
```

Local printers with RS-232C or PARALLEL interfaces will **not** be recognized by the standard TABLE program. If option `RS232` is chosen, you must have an HP 98626 or 98644 RS-232C Serial interface or an HP 98628 Datacomm interface present. In order to use one of these Serial interfaces, module RS232 must be installed; using the Datacomm interface requires module DATA_COMM.

Here are the default interface switch settings:

- Select code 9
- Interrupt Level set to 3
- Baud rate set for 2400 baud
- Stop bits set for 1 stop bit
- Bits/character set for 8 bits
- Protocol set for XON/XOFF
- Parity set to off

If your printer uses a different parameter, then you should change the interface switch setting[1] accordingly; see the interface's installation manual for switch locations and settings. If you want to use another select code, then you will need to modify the `local_RS232_printer_default_dav` parameter accordingly; see the subsequent discussion of Default Device Address Vectors.

If the option PARALLEL is chosen, a Series 300 SPU that provides a built-in parallel interface must be present. In order to use this interface, the PARALLEL module must be installed.

Local Printer Timeouts

This constant determines the timeout parameter for local printers:

```
const
  local_printer_timeout =
    $IF local_printer_option=HPIB$
       12000;  {milliseconds}
    $END$
    $IF local_printer_option=RS232$
       0 ;     {infinite}
    $END$

    $IF local_printer_option=PARALLEL$
       10000;  {milliseconds}
    $END$
```

This governs the byte-transfer timeout used by the local printer driver. The timeout, expressed in milliseconds, specifies the maximum time allowed for each byte handshake to complete. A value of zero is a special case, specifying an infinite timeout. See the commentary above this constant declaration in the CTABLE.TEXT source program for recommended values.

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

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

[1] With the 98644 card, you may need to write a short application program to set the parameters. See the *Pascal Procedure Library* manual for details.

Default Device Address Vectors

Here are the default "device address vectors" for devices that cannot be found by interrogation.

```
{default dav's for devices not found by scanning}
  type dav_type = {device address vector}
    packed record
      sc, ba, du, dv: -128..127;
    end;
  const
    HP9885_default_dav =
      dav_type[sc:12, ba:-1, du:0, dv:-1];
    SRM_default_dav =
      dav_type[sc: 21, ba: {node} 0,
               du: {unit} 8, dv: -1];
    BUBBLE_default_dav =
      dav_type[sc: 30, ba: 0, du:  0, dv:  0];
    local_HPIB_printer_default_dav =
      dav_type[sc:  7, ba: 1, du: -1, dv: -1];
    local_RS232_printer_default_dav =
      dav_type[sc:  9, ba: 0, du: -1, dv: -1];
    local_PARALLEL_printer_default_dav =
      dav_type[sc: 23, ba: 0, du: -1, dv: -1];
```

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

- `sc` is the interface select code. Select code 7 corresponds to the built-in HP-IB port at the back of Series 200/300 computers. The HP 9885 disc is connected using a 16-bit parallel interface on select code 12, and a DMA card. The SRM interface is normally set to select code 21.
- `ba` is the HP-IB primary address of the peripheral. Usually an 8290x is addressed as device 0; so is a 9895. The 913x family of hard discs are expected (though not required) to be at primary address 3, and printers at address 1.

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

  For a SCSI interface, `ba` indicates the SCSI device number (0-7).
- `du` selects the disc unit in a multi-drive machine. For instance, a 9121D has drives 0 and 1. With SRM systems that contain multiple disc drives, this parameter selects which disc is to be accessed.
- `dv` selects a particular volume in a multi-volume CS80 (79xx family) disc.

Hard Disc Partitioning

The next section that you will come to in the CTABLE.TEXT source program concerns hard disc partitioning. However, in order to be able to wisely decide whether or not you will need to modify any of these parameters (and, if so, to choose which parameter you want), you need to fully understand how partitioning works.

Note

If you haven't read the discussion of hard disc partitioning in The Booting Process discussion, you should do so now.

The standard TABLE program assumes that non-HFS local hard discs are to be partitioned into several "logical volumes," each of which is to be assigned a unit number. The following equation conceptually describes how TABLE determines the number of volumes into which the disc is to be logically divided (the equation actually used is slightly more complex, because it partitions on track boundaries):

`nvols` = disc capacity DIV `mvs`

The values of `nvols` for each type of hard disc, as calculated by this equation, are shown near the end of the main part of the CTABLE program, following the comment:

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

The `mvs` parameter is a constant in the `options` module.

```
{local hard disc partitioning parameters}
  type
    pp_type = {partitioning parameters}
      record
        mvs: integer;    {min vol size in bytes}
        mnv: shortint;   {max number of volumes}
      end;
```

There are comments in the program about values and effects of `mnv` parameter.

```
  const
    min_size = {in bytes [1..maxint]}
      1000000;
    max_vols = {[-30..30]; <0 means auto-coalesce}
      -30;
    HP913X_A_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    HP913X_B_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    HP913X_C_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    CS80disc_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    SCSIdisc_pp =
      pp_type[mvs: min_size, mnv: max_vols];
```

The constant `min_size` indicates that no logical volume is to be smaller than one million bytes. The constant `max_vols` indicates that no device is ever to be partitioned into more than 30 logical volumes; the negative value of `max_vols` indicates that logical volumes that do not have valid directories are to be "coalesced" with the last preceding volume found to have a valid directory. These constants are assigned to the `mvs` and `mnv` constants for each class of device. You can change them if desired; the values and corresponding effects of `mnv` are described in the comments in the CTABLE program.

Note

The constant `max_vols` must **not** be greater than 30.

The `HP913X_A` corresponds to all 5-Mbyte HP 913x Option 10 "A" drives and "V" drives with a single "disc unit" or "drive number" (as opposed to non-Option 10 "A" drives which have 4 drive numbers).

The `HP913X_C` corresponds to all 15-Mbyte HP 913X "XV" drives.

Example of Standard Partitioning

In order to better understand partitioning, let's look at how the standard TABLE program partitions an HP 7908 hard disc. You can see most of the default parameters by looking in the `templates` section of the main program that begins with this comment:

```
$if false$ { current CS/80 discs "soft" partitioned by the host }
```

These discs have a capacity of about 16 Mbytes, so `nvols` is 16 for this type of disc.

The size of each logical volume is given by this equation:

```
vol_bytes = tpm DIV nvols * bpt
```

in which:

`vol_bytes` = size of volume (in bytes)
`tpm` = number of tracks per disc media (device-dependent)
`nvols` = number of volumes expected on the disc (device-dependent)
`bpt` = bytes per track (device-dependent)

The last volume on the disc may contain some additional bytes according to the remainder of the above integer division:

Last volume = `vol_bytes` + (`tpm` MOD `nvols`) * `bpt`

Here are the values of the preceding parameters for an HP 7908 disc drive (they are contained in the main body of the program, near the end):

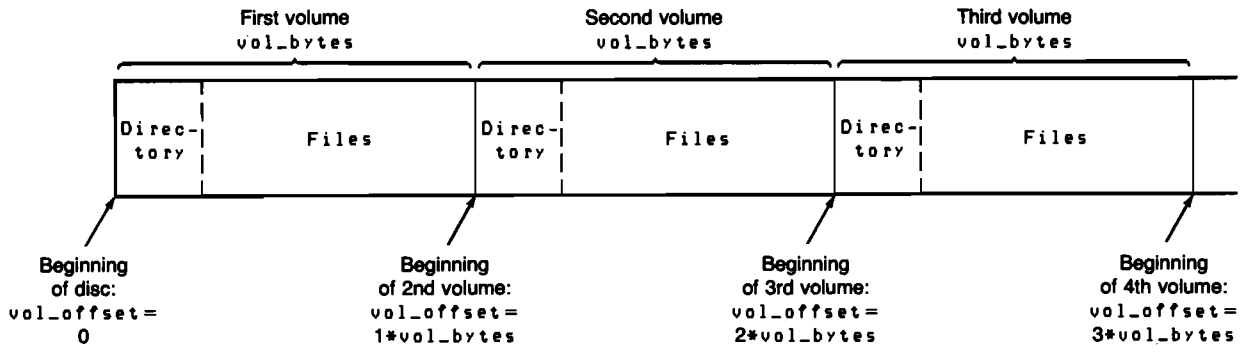
```
tpm   =  5 * 370    { 5 surfaces with 370 tracks/surface }
nvols = 16
bpt   = 35 * 256    { 35 sectors/track with 256 bytes/sector }
```

Therefore:

```
vol_bytes  = ((5 * 370) DIV 16) * 35 * 256 = 1030400 (bytes)
```

The `tpm`, `bpt`, and `nvols` parameters for 913x hard discs are found in the `medium_parameters` function in module `ctr`.

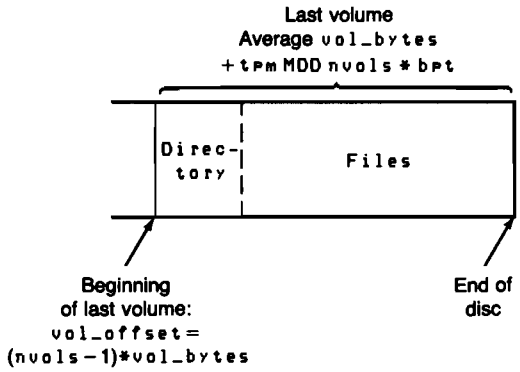
Here is a diagram of how the TABLE program partitions non-HFS hard discs:



The first directory is placed at the "beginning" of the disc (at an offset of 0 bytes on track 0). The data area used for files immediately follows the directory.

The next directory is placed so as to just follow the end of the first volume. The size of the first volume determines the actual location where the second logical volume will begin. This rule is also followed by each successive logical volume on the disc.

The last volume on the disc looks as follows:



When TABLE attempts to validate unit numbers, it looks at these logical volume boundaries in search of valid directories. As each valid directory is found, the corresponding volume is assigned a unit number. If a valid directory is not found at its expected location, then the corresponding unit is invalid; the amount of disc space normally occupied by this volume can be coalesced with the last preceding logical volume found to have a valid directory, if desired.

Partitioning Recommendations

Here are the general recommendations as to how you can change the standard partitioning of hard discs:

1. The simplest method of changing the partitioning on your hard discs is to "coalesce" adjacent logical volumes. You should try to use this solution if possible.
2. If coalescing does not provide you with an adequate solution, you can also set up your own logical volume structure on the disc by modifying the parameters in the CTABLE source program.

a. The easiest changes might be to change the `nvols` parameter in the `templates` section that corresponds to your disc. For instance, changing this constant from 30 to 15 for the 7912 allows you to have two 7912 drives automatically assigned unit numbers by CTABLE. The size of the logical volumes will be doubled, and partitioning will still be made on track boundaries. Changing `cs80disc_pp_mnv` (in module OPTIONS) to 15 may also suit your needs.

b. If the above methods still do not provide an adequate solution, read the subsequent discussion in Designing Your Own Partitioning Schemes.

Coalescing adjacent hard disc volumes was discussed earlier. Modifying the TABLE Program is discussed momentarily.

Note

If you do modify the standard TABLE program, keep in mind that you must use a version of the program that uses the *same* partitioning scheme in order for all logical volumes to be recognized properly.

Designing Your Own Partitioning Schemes

The recommended method is the standard TABLE partitioning method. Here is the section of the `template` (toward the end of the main CTABLE program) that performs the standard partitioning:

```
for i := 0 to nvols-1 do
  if not hfs_installed then
    tea_CS80_mv(11+i, primary_dam, {sc} 7, {ba} 0, {du} 0, {dv} 0,
                      vol_offset(i, nvols, mp),
                      {devid} CS80id,
                      vol_bytes(i, nvols, mp),
                      mp.tmp*mp.bpt);
```

The `vol_offset` and `vol_bytes` functions calculate the offset and size of each of your directories according to the `nvols` and `mp` values; you can use the standard values, provided in this same template, or specify your own. If, for example, you wanted to change only the value of `nvols` for a 7908 disc to 8, you could change this line in the template (just a few lines before the standard partitioning algorithm shown above):

```
CS80id := 7908; nvols := 16; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

to this:

```
CS80id := 7908; nvols :=  8; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

The `vol_offset` and `vol_bytes` functions would then make the volume offset and size calculations for you.

The last argument in this call is new for Pascal 3.2. It corresponds to a parameter for disc total size in the tea procedure `tea_CS80_mv`, and is used to claim the whole disc if HFS is found on the disc.

While the standard method is the one recommended, there is nothing that prevents you from using your own. If you like, you may remove the `for` statement, duplicate the `tea` procedure call `n` times, and specify volume offsets and sizes of your choosing for each logical volume. Here is an example of one for unit 11 (you will have to supply actual values of the example parameters `offset_for_unit_11` and `bytes_for_unit_11` shown below):

```
tea_CS80_mv(11, primary_dam, 7, 0, 0,
                  offset_for_unit_11,
                  CS80id,
                  bytes_for_unit_11,
                  mp.tmp*mp.bpt);
```

The `tea` procedure checks to ensure that your logical volumes each lie inside the media boundaries. Unfortunately, the `tea` procedure doesn't check to see if any of them overlap!

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

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

This allows you to use the standard partitioning method, except that the partitioning will occur on 512-byte block boundaries — not necessarily on track boundaries. The "{" character at the beginning of the line makes the line a comment; enable compilation of the line by deleting the "{" character. Depending upon the media parameters and the number of logical volumes, this may or may not make a difference in how your media actually gets partitioned. This feature is provided solely for compatibility with discs used with Pascal 1.0. If you don't need it for this reason, don't use it!

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

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

If you still can't find the source of the error, you can re-compile CTABLE with $DEBUG ON$. Get a listing from the Compiler, too. Making sure that the DEBUGGER (found on ASM: disc) is installed, execute CTABLE again. When it terminates with the error again, use the queue (Q) command in the Debugger to determine the line numbers of the statements leading up to the error. Also, when you examine the queue, you may need to trace back several line numbers to actually locate the offending statement.

System Unit Auto-Search Declarations

These constants determine the order of devices searched while trying to find a system volume.

```
{system unit auto-search declarations}
  const
    sysunit_list_length =
      7;
  type
    sysunit_list_type =
      array [1..sysunit_list_length] of unitnum;
  const
    sysunit_list =
      sysunit_list_type[
      first_harddisc_lun, {first hard disc logical unit number}
      45,   {srm, prefixed to user's sysvol}
      4,    {floppy unit 1, primary dam}
      44,   {floppy unit 1, secondary dam}
      3,    {floppy unit 0, primary dam}
      43,   {floppy unit 0, secondary dam}
      42];  {bubble}
```

If a valid directory is not found on any of these units, then the system volume is determined by the normal algorithm (described in The System Volume section of The Booting Process discussion presented earlier this chapter).

If a system unit was explicitly specified by modifying the constant called `specified_system_unit` at the beginning of the module called `options`, then this search will not override the specified system unit.

HP-IB Select Codes Searched

These constants determine the select codes scanned in search of an HP-IB type interface, including 98625 High Speed Disc interfaces.

```
{HP-IB select code scanning declarations}
  const
    sc_list_length =
      3;
  type
    sc_list_type =
      array [1..sc_list_length] of shortint;
  const
    sc_list =
      sc_list_type [
        7,      {internal HP-IB}
        8,      {default sc for HP98624 HP-IB}
        14];    {default sc for HP98625 HP-IB}
```

The select codes are searched in the order they appear in the list (7 first). On each select code, addresses 0 through 7 are polled in succession for devices. In the case of multiple devices contending for an assignment class, say multiple local hard discs where the total capacity of all is greater than 30 Mbytes, generally the last one polled will be the one assigned a logical unit number.

SCSI Select Codes Searched

These constants determine the select codes scanned in search of a SCSI type interface.

```
{SCSI select code scanning declarations}
  const
    SCSIsc_list_length =
      3;
  type
    SCSIsc_list_type =
      array[1..SCSIsc_list_length] of shortint;
  const
    SCSIsc_list =
      SCSIsc_list_type[
          14,    {default sc for HP98265A (internal) and HP98658A (external)}
          15,    {external SCSI sc when internal HP-IB/SCSI present at sc 14}
          28];   {internal SCSI on 340}
```

The select codes are searched in the order they appear in the list (14 first). On each select code, an INQUIRY command is sent to devices 0 - 7. Responding devices that are of a type supported by the Pascal Workstation are remembered and an attempt is made to assign a unit number. When multiple discs that use a LIF directory structure with a total capacity of more than 30 Mbytes are encountered, the last device found will be the one assigned a logical unit number.

SCSI Removable Disc Options

SCSI removable media may be an optical disk which has capacities of greater than 300 megabytes. Therefore, removable media can be configured to be:

1: A Hard disk if it has a size greater than 10M.

2: A Hard disk always.

3: A Floppy disk always.

by setting the `SCSIRemovableOption` constant to an appropriate type. CTABLE provides the following types and constants:

```
type
     SCSIRemovableOptionsType = (AllOver10MAreHard,
                                 AllAreHard,
                                 AllAreFloppy);
const
     SCSIRemovableOption = AllOver10MAreHard;
```

When SCSI removable media is being treated like a hard disk, be aware of these facts:

1: If the removable media is not online at the time CTABLE is executed, the size of the disk is not available. If the `AllAreHard` option is being used, then a unit entry will NOT be created for it. If the `AllOver10MAreHard` option is used, then a unit entry for a floppy disk will be created for it.

2: CTABLE will attempt the PREVENT MEDIUM REMOVAL command. Through the SCSI Programmer's Interface the ALLOW MEDIUM REMOVAL command may be sent.

3: If the removable media goes off line for any reason, such as removing the media, PWS will discontinue communication with that device until CTABLE has been rerun.

About Module CTR

This module should not be modified!

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

Each peripheral is assigned a letter designator; these are listed in the export section of module `ctr`. In addition there is descriptive information about the size of each type of device, expressed in bytes per track and tracks per medium. The routines in `ctr` avoid partitioning across track boundaries, which would cause very inefficient disc access patterns.

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

There are also some utility routines. The `create_temp_unitable` procedure allocates in the heap a temporary structure of the same type as the real system Unit Table. CTABLE makes its assignments to this temporary structure, then uses `assign_temp_unitable` to copy the final result into the actual system table. Note that `assign_temp_unitable` will not overwrite any RAM volumes which have been created in the system unit table. This feature is provided so that if you execute a CTABLE while the system is running, you won't lose files in memory.

The `sysunit_ok` function checks to see if a particular unit is blocked, on-line, and has a valid directory; if so, it is a legal candidate for the system unit.

If you look at the assignments to the various fields of a Unit Table entry, you will be aware that two of them are procedure variables which must be initialized to the names of the DAM (Directory Access Method) and TM (Transfer Method or driver) appropriate to the volume and physical device. DAMs and TMs are not part of CTABLE and so would ordinarily be linked to modules already in RAM by the linking loader when CTABLE is loaded.

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

About Module BRSTUFF

This is another module which shouldn't be modified!

It exports two routines. The function `internal_mini_present` determines if there are any internal flexible disc drives in your computer. The function `get_bootdevice_parms` determines what type of device was used for booting and returns the `dav` (device address vector) for that device.

About Module SCANSTUFF

This module shouldn't be modified!

Its purpose is to interrogate certain HP-IB disc drives about their size and identification. To do this, the `value` routine (see module `ctr`) is used to find routines which are present only if the driver modules supporting these discs are installed.

About Module SCSIscanstuff

This module should not be modified.

Its purpose is to allow interrogation of:

- Select codes for SCSI interfaces
- SCSI interfaces for SCSI devices at particular addresses
- SCSI devices for identification and size information.

To implement the items in the above list, the `value` routine (see the `ctr` module) is used to determine if the SCSI disc driver (SCSIDISC) is installed.

Discussion of the Main Body of CTABLE

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

Default DAV Assignments

The program first assigns default device address vectors (DAVs) for devices that cannot be found by scanning (such as printers and HP 9885 8-inch disc drives).

HP-IB Interfaces Scanned

After various initializations, CTABLE scans the HP-IB select codes listed in module options. For each HP-IB interface found, and for bus addresses 0 through 7 on each interface, the program inquires to see if a device is present. A letter designating the device is returned. You can see the definitions of these letters in the constant declaration at the beginning of the `ctr` module.

SCSI Interfaces Scanned

The CTABLE program scans the SCSI select code listed in module options. For each SCSI interface found, and for device 0 through 7 on each interface, the CTABLE program inquires to see if a device is present. A letter designating the device type is returned. You can see the definitions of these letters in the constant declarations at the beginning of the `ctr` module.

Boot Device Info

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

Temporary Unit Table

A temporary Unit Table is then created in the heap. The assignments made as CTABLE executes will be made to elements of this temporary table; only at the end will the real system Unit Table be updated.

Standard Assignments

Next, "standard" unit number assignments are made. It is wise not to change these assignments, since programs tend to depend on them.

- Unit #1 is assigned to the screen (CONSOLE:)
- Unit #2 is assigned to the keyboard (SYSTERM:)
- Units #3 and #4 will be assigned to the highest priority flexible disc drive. If both internal drive(s) and an external flexible disc drive are present, the internal drive(s) will be used for #3 and #4 unless the external disc was the boot device. This policy gives preference to the higher-performance internal floppy disc drives.
- If an SRM interface is present, it is assigned unit #5. (It may also be assigned unit #45 later in the program.)
- Unit #6 is assigned to the local printer (PRINTER:). This assignment is made whether or not a printer is actually connected to the computer, because there is no way to interrogate every possible type of printer.

Additional Floppy Unit Pairs

Next, the second and third pair of flexible disc drives are assigned unit numbers. Units 7 and 8 are assigned to the second highest priority floppy drive pair, and 9 and 10 to the third priority pair.

Multiple Local Hard Discs

With auto-configuration, CTABLE can deal with several local hard discs found during the HP-IB and SCSI scanning process (previous versions of this program, without modification, could only find one). This code is surrounded by conditional Compiler options, because you may wish to not compile it and instead force particular assignments.

CTABLE will break a hard disc (which has not previously been initialized to a single volume, and is not of HFS format) into multiple volumes. As things are arranged (see module `options`), no volume will be less than one million bytes and no disc will be divided into more than 30 volumes. The units assigned to these volumes begin with #11 and can use up through #40, depending on the number required for each disc.

Discs will not be split into multiple volumes if the HFS_DAM module is in INITLIB and the disc contains an HFS superblock. If the module is missing, the disc may be incorrectly shown as having multiple volumes. This situation is dangerous if data on the HFS is needed. It should be rectified immediately, else HFS files may be destroyed.

Tape Cartridge Drives

If there are any stand-alone tape cartridge drives present, or any CS80 disc drives with integrated tape drives present, then the program also finds them. The highest priority tape is assigned unit number 41, and the second priority tape is assigned unit number 42.

Alternate DAMs

Next, the alternate-DAM entries are assigned. This allows all flexible discs to be used if their directories are of either primary_dam or secondary_dam type. Units #43: and #44: are alternates for #3: and #4:. For instance, if LIF is the primary DAM, then units #43: and #44: will use the alternate DAM, HFS DAM (or UCSD, depending on value of `thisversion`) to access drives corresponding to #3: and #4:. (Alternates for units #7: thru #10: are a few lines later in the program.)

Duplicate SRM Unit Entries

The "duplicate entries for prefixing down the SRM" section provides templates that you can use to assign additional unit numbers to SRM directories. For instance, suppose you want to have unit #46: assigned to the directory called `/SPECIAL/USER10/FRED`. Enable the first template by deleting the { comment brace preceding it. Then scroll down until you find the comment `{ prefix the primary and secondary SRM unit entries }`. (It may be easier to use the Editor's Find command, since these templates are a couple of pages away from the first templates.) Enable the template for #46: by deleting the { comment brace, and replace the ? with the desired directory path `SPECIAL/USER10/FRED`.

Note that as of version 3.2, #46 may be used automatically as the boot or system unit so do not assign #46: if you are booting from an HFS hard disc.

#45 is not really an alternate; it is another SRM volume, and may be assigned as the system volume later. If this happens, the operating system will have two units on the SRM: one for the "system volume," which is used for temporary system files, work files, stream files etc.; and another for the "default" working directory. This avoids any possible need to prefix an SRM system volume away from where it should be.

More Alternate DAMs

Next, units #47: and #48: are assigned as alternate DAM units for #7: and #8: (second priority floppy disc pair). Units #49 and #50 are alternates for #9: and #10: (third priority floppy disc pair).

Templates

Next are the "templates" for overriding the mass storage table entry assignments made by the standard TABLE. These templates are surrounded by conditional `$if false$` Compiler options which cause them to be skipped. Thus, the `tea` procedure calls have no effect until you change the `$if false$` to `$if true$`. The `tea` procedures themselves, are defined in the module `ctr`. They actually perform the Table Entry Assignments.

There are templates for the following disc drives: internal; 8290x (Amigo); 9895; 913xA, B, V, and XV; CS80 hard discs (HP 7908, 7911, 7912, 7914, 7933, and 7935); SS80 flexible discs (such as the 9122) CS80 tapes; Magnetic Bubble memory cards; EPROM cards. Each template gives the opportunity to specify the following:

- unit number
- directory access method (DAM)
- select code
- bus address (HP-IB interfaces)
- drive unit

And for some types of devices:

- offset in bytes from beginning of volume to this unit's directory (for "soft-volume" discs)
- drive type (the variable named letter in a constant declaration of module `ctr`)
- total size of disc (for "soft-volumed" discs)

For multiple-volume drives, the templates include a `for` loop which calculates how to break up the disc space in the preferred fashion.

If you want to change the default for an HP 9121 drive, you will need to use the `tea_HP8290X` procedure. The reason for this is that the HP 9121 drives behave just like the HP 8290X drives. You might also note that you would also use the `tea_HP8290X` procedure for the 5.25-inch drive in the HP 9135 and the 3.5-inch drive in the HP 9133.

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

```
type
  ds_type =  {Directory access method Specifier for local mass storage}
    ( primary_dam,      {normally LIF }
      secondary_dam,    {HFS or UCSD, depending on choice in options}
      LIF_dam,          {LIF, regardless of primary/secondary choice}
      UCSD_dam,         {UCSD, regardless of primary/secondary choice}
      HFS_dam   );      {HFS, regardless of primary/secondary choice}
```

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

The remaining parameters for all the local mass storage `tea` procedures are device-specific. Most devices will need addressing information such as select code (`sc`), HP-IB bus address (`ba`), and disc unit number (`du`).

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

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

Place all tea procedure calls here.

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

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

Temporary Unit Table Copied

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

SRM Prefix Directories

The SRM unit entries are then prefixed to the appropriate directories. Each workstation in an SRM system has an identification number called its "node number", and it is **strongly** recommended that the system be configured so that every workstation's node number is unique. #5: is prefixed to the root directory "/" if possible. You can change the working directory to your own directory by adding the directory path to the slash (/).

CTABLE tries to prefix #45 to a directory called /WORKSTATIONS/SYSTEMnn, where nn is the node number. If no such directory exists, it tries to use directory /WORKSTATIONS/SYSTEM (with no node number). If that one doesn't exist, entry #45 is nullified. This is a rather key mechanism. It allows the workstations in an SRM system to each have unique configurations. For the normal functioning of the Pascal system, a system volume is required to hold the system library and various system files. If all workstations shared the same system volume, file name collisions would be a real nuisance. CTABLE supports this partitioning, and so does the overall booting process, allowing for instance a different INITLIB and TABLE for each workstation.

Remove Extraneous Hard Disc Volumes

When a valid directory is not found at the expected location on the disc, then the corresponding unit number is made invalid. This service is performed by the section of code in the main part of CTABLE that follows the comment:

```
{ remove extraneous local hard disc entries if necessary }.
```

If desired, the volumes which don't have valid directories may be "coalesced" with the last valid directory found which precedes this invalid directory. See the section "Coalescing Hard Disc Volumes" presented earlier in this chapter.

System Unit Selected

The system unit is then selected according to the priorities set in the constant called `sysunit_list`, exported from module `options`.

SRM System Unit Selected

If the system unit is #45: (SRM system volume), then unit #5 is also an SRM volume. In that case, #5 is set up as the initial default volume for the system right after it boots up.

HFS System Volume Selected

If the boot unit was a non-removable HFS unit, then unit #46: is prefixed to `/WORKSTATIONS/SYSTEMxxx`, where `xxx` corresponds to the boot file name of `SYSTEM_XXX` or `SYSxxx`. If this fails, an attempt is made to prefix #46: to `/WORKSTATIONS/SYSTEM` on the system unit. If both fail, unit #46: is not assigned.

Extra HFS Unit Entries

A block of comments in the code explains how to create extra units for HFS discs. The last comment line shows how to call `extra_HFS_unit` to set up an additional unit entry to the HFS disc at unit #11, and prefix that entry to `/PROGS`. Note that if you had two HFS hard discs, one would appear as #11, and the other as #12. You might add extra HFS units for each at #21 and #22 as follows:

```
extra_HFS_unit(11,21,'/MYDIR');
extra_HFS_unit(12,22,'/USERS');
```

Here, 11 and 12 are the **base unit numbers** for these discs, and should be used as the first parameter in the `extra_HFS_unit` call.

System Files Re-Opened

This procedure re-opens the standard unblocked system 'files':

- #1: is assigned to SYSTERM:
- #2: is assigned to CONSOLE:
- #6: is assigned to PRINTER:

Editing CTABLE

If you have just read through the preceding discussion for the first time, you will need to go back and read the relevant sections and make the desired changes.

If you have already edited the CTABLE source program, you are ready to store your new file. Quit editing and Write the edited CTABLE in a new file, such as NEWCTABLE (or use the Save option if you are editing a backup copy of the file). Exit the Editor by typing [E].

Compiling and Running CTABLE

1. The modules in CTABLE.TEXT import modules from INITLIB. However, the interface text for these modules is not available unless you enable the `$search 'CONFIG:INTERFACE'$` (`$search 'ACCESS:INTERFACE'$` if your software was purchased on double-sided media) Compiler option at the beginning of the source program (by removing the comments from the appropriate line). You must also be sure that this disc is on-line during the compilation of the CTABLE program; you could also copy the file onto another on-line disc and change the volume specification in the program accordingly.
2. Load the Compiler by typing [C] (you may need to put the CMP: disc on-line). Answer the Compiler's `Compile what text ?` prompt by entering:

   ```
   NEWCTABLE
   ```

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

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

4. Press [Enter] (or [Return]) to say that the default output file name of "`SYSVOL:NEWCTABLE.CODE`" is fine.

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

5. Press [R] or **RUN** to execute NEWCTABLE.

Verifying the New Configuration

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

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

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

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

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

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

Making the New Configuration Permanent

Once you are satisfied with your new configuration and wish to make it permanent (i.e., it will be set up each time you boot unless you change it again), copy the code file to your BOOT: (or BOOT2:) disc. First, however, you should link the new file to itself in order to conserve disc space.

Link the Modules Together

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

   ```
   NEWCTABLE
   ```

3. To conserve space on the disc, you can specify a header size smaller than the default (38). Press [H], and enter: 1. The header size is then changed to the minimum (18).
4. Press [O] (for Output) and enter:

   ```
   NEWCTABLE
   ```

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

Now you are ready to perform the final operations.

Copying the BOOT: Disc

To be safe, make a copy of your current boot disc (BOOT: or whatever). If all goes well, you will not need it. However, if something goes wrong, you can always try again **if you make a backup copy of your boot disc.**

Assuming you have two drives, put the current boot disc in unit #3 and an initialized blank disc in unit #4. Use the Filer to Filecopy #3: to #4:. Only one drive? You will need to Filecopy #3: to #3: and swap the discs when prompted.

After you have made a backup copy, install the new TABLE by following the directions given next.

Install the New TABLE

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

   ```
   BOOT:TABLE
   ```

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

   ```
   BOOT:
   ```

4. Respond to `Crunch directory BOOT: ? (Y/N)` with [Y].
5. Now copy the new code file from SYSVOL: to BOOT:, giving it the required name. Insert the SYSVOL: disc, press [F] (for Filecopy) and enter:

   ```
   NEWCTABLE.CODE,BOOT:TABLE
   ```

6. Swap discs as directed by the Filer.
7. Save your new source file on the CONFIG: disc too. Insert the SYSVOL: disc, press [F] and enter:

   ```
   NEWCTABLE.TEXT,CONFIG:$
   ```

8. Swap discs as directed by the FILER.
9. Clean up the SYSVOL: disc by removing all the files you put there. Use wildcards to save typing. Insert the SYSVOL: disc, press [R], and enter the ? wildcard.
10. Respond [N] to the prompt to remove LIBRARY, and respond [Y] to the prompts to remove INTERFACE, NEWCTABLE.TEXT, and NEWCTABLE.CODE. Respond [Y] to the confirmation prompt.
11. Exit the FILER by typing [Q].

Keep the copy of NEWCTABLE.TEXT somewhere safe! This may save you a lot of work should your configuration change, or your new BOOT: disc become damaged or lost.

Setting Up an SRM System

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like disc drives, printer spoolers, and plotter spoolers. Also, the SRM can be the only mass storage device for a machine with no local disc drives.

This section only gives a rough outline of what is required to configure Pascal workstations in order to access an SRM system. Here are the main steps:

1. Add modules DATA_COMM and SRM to INITLIB and re-boot, or execute them and re-execute TABLE; this step provides minimal access to the SRM through unit #5:
2. Copy files to certain SRM directories, and optionally re-name files; this step allows you to use unit #45: as the system volume and to boot from an SRM (if your computer is equipped with Boot ROM 3.0 or later)
3. Modify the TABLE program, and re-execute it; this step allows you to assign additional unit numbers to the SRM system

Configuring a Pascal workstation to access an SRM system is covered in more detail in the SRM documentation supplied with the SRM product and in the *Pascal User's Guide.*

Example SRM Configuration

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like discs, printers, and plotters. Also, the SRM may be the only mass storage device for a machine with no local disc drives.

This section explains how to configure workstations to access and boot Pascal 3.2 from an SRM system. It is used as the example "custom" configuration because it can employ three methods of modifying the standard configuration:

- Copying and re-naming files
- Adding modules to INITLIB
- Modifying the TABLE program (optional).

This section tells what to do the *first* time you set up the *first* Pascal workstation to access an SRM system. It should not be repeated for every workstation you set up. Once this procedure is complete, the SRM will be accessible any time you boot up your workstation.

Prerequisites

Here are the assumptions made by this set-up procedure.

Who Should Set Up the SRM

The person who is designated as the "SRM system administrator" should perform the process described in the next few pages. Refer to the SRM *Software Installation Manual* for more information.

SRM Hardware

It is assumed that your SRM hardware has been installed and tested as prescribed in the SRM documentation. In order for your system to work with the SRM, every workstation in the SRM configuration must have a unique node number (see the SRM System Manual to learn about node numbers). You will also need the wiring chart and node number assignments which were prepared when designing and installing your SRM system.

Boot ROM Versions

If you have an HP 9816 Computer with a Boot ROM 3.0L, then you must boot from a local disc drive. The SRM can only be used after normal booting is complete. Similarly, if you have an HP 9826 or 9836 Computer with a Boot ROM with version number less than 3.0, then you must boot from the internal 5.25-inch flexible disc drive. In both of these cases, you will probably want to make a back-up copy of the original BOOT: disc, as you will be modifying the INITLIB file on that disc.

If your computer is equipped with Boot ROM 3.0 or later version, it is possible to boot directly from the SRM. System Boot files are found on the SRM system in the /SYSTEMS root directory; they have names like SYSTEM_P. The other files used at boot time (INITLIB, STARTUP, and TABLE) are found in the /WORKSTATIONS/SYSTEM directory. This too is explained in the SRM System Manual.

Overview of SRM Installation

Configuring your system to access SRM is not a hard or complicated operation, but it is important that you follow the subsequent procedures in exact detail. Since you are less likely to make mistakes if you understand what's going on, here is an outline of what you will do.

1. Install driver modules DATA_COMM and SRM by executing them (they are actually programs that install themselves automatically).
2. Execute the TABLE auto-configuration program. When it is executed while the DATA_COMM and SRM driver modules are installed, it will find the SRM system and assign unit #5 to the SRM.
3. If they are not already on the SRM system, create directories /SYSTEMS and /WORKSTATIONS/SYSTEM.
4. Copy the system Boot file (SYSTEM_P) to the /SYSTEMS directory. Copy the rest of the Pascal system files to the /WORKSTATIONS/SYSTEM directory. (The Boot ROM expects to find the Pascal system in these directories.)

5. Use the Librarian to create (on the SRM) a new INITLIB file that contains the additional modules DATA_COMM and SRM, and then replace the existing INITLIB with this new one. (If you have Boot ROM 3.0 or later, then you will be replacing the INITLIB in the /WORKSTATIONS/SYSTEM directory; with earlier Boot ROMs and Boot ROM 3.0L, you will be replacing the INITLIB on the BOOT: disc.)
6. Re-boot the computer, and verify the new configuration.
7. You can also optionally modify the TABLE program to assign additional unit numbers to the SRM system.

Installing the SRM Driver Modules

First, install the DATA_COMM module (from the CONFIG: or ACCESS: system disc). Although you may have already copied the file onto another volume, such as a local hard disc, this example assumes that you will be loading and executing it from the CONFIG: disc.

Execute the file by pressing [X] at the Main Command Level. The system prompts:

```
Execute what file?
```

Enter this file specification:

```
CONFIG:DATA_COMM.
```

Be sure to include the trailing period to suppress the ".CODE" suffix.

Install the SRM module similarly; it is also on the CONFIG: disc as shipped to you.

Re-Configuring with TABLE

Use the eXecute command to execute the TABLE program; it is on the BOOT: disc supplied with your system. Press [X], and then answer the `Execute what file?` prompt with this file specification:

```
BOOT:TABLE.
```

Again, be sure to include the trailing period.

When the program has finished, you can use the Filer's Volumes command to see that unit #5 is assigned to the SRM system. From the Main Command Level, press [F] and then [V]. Here is a typical display:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 # BOOT:
  5 # ROOT:
  6   PRINTER:
```

If the name of the SRM's root directory is not shown in the display, re-execute all three programs (DATA_COMM, SRM, and TABLE). You may have done something wrong in that process.

If the Filer's Volumes command still does not recognize the #5: volume, check to see whether the SRM hardware is properly configured and installed. For instance, the (unmodified) TABLE program expects that the SRM interface in your computer is set to select code 21.

If that does not work, then you should refer to the troubleshooting sections of the SRM System Manual.

Creating the Required Directories and Files

The first time that a workstation is set-up to access an SRM system, you will need to set up certain directories on the SRM. These directories have special functions, as described in the following paragraphs.

A Sketch of Normal SRM Directory Configuration

In order to allow each Workstation in an SRM configuration to boot up a unique system and have its own system volume, a private directory is established for each node number.

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

In the SRM's root directory there should be another directory called WORKSTATIONS. Under this there should be a directory called SYSTEM, and for each node number "nn" there should also be a directory called SYSTEMnn. For instance, if there are three Workstations on nodes 08, 14, and 15, then the following directories should exist in the root:

```
WORKSTATIONS/SYSTEM
WORKSTATIONS/SYSTEM08
WORKSTATIONS/SYSTEM14
WORKSTATIONS/SYSTEM15
```

Under WORKSTATIONS/SYSTEM should be copies of all the system files, such as the Compiler, Filer, and Editor.

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

Also in a node's private system directory can be the files which "personalize" a Workstation: customized copies of LIBRARY, INITLIB, TABLE, AUTOSTART, CTABLE.TEXT and so forth. (Personalized copies should be separate files and not duplicate links to one file.)

Once this set-up is created, booting is a smooth and automatic process. With Boot ROM 3.0 and later versions (but not 3.0L), you can boot from the SRM; the particular system to be booted is selected by name at power-up. Thereafter, the Workstation looks for the necessary files in the directory with its node number. If INITLIB can't be found in the /WORKSTATIONS/SYSTEMnn directory, default is taken to /WORKSTATIONS/SYSTEM; if something crucial is still missing, the boot may fail. (The computer will complain to the operator.)

If you boot from the SRM or if you have no local hard disc on-line, your system volume will be unit #45 (prefixed to your private directory /WORKSTATIONS/SYSTEMnn) and your default volume will be #5 (another SRM volume, prefixed to the SRM root directory). Even if the SRM is not chosen as your system volume (using the scheme above), it will still be accessible through units #5 and #45.

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

Most users will also want a private directory for their default volume. Typically one creates a directory called USERS under the root, and within USERS a private directory for each individual. After booting, use the Filer to set the current working directory for your unit #5 to your private directory (you can modify the TABLE program or create an AUTOSTART file to do this for you). This keeps the root directory from getting cluttered.

Setting Up SRM Directories

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

If the SRM has already been running with some other systems connected, such as an HP 9845 or 9836 running BASIC, some of these directories may already exist. To see the directories which already exist, press [L] for the List directory command, and enter the root-level directory specification:

```
#5:/
```

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

To create directory /WORKSTATIONS, use the following Filer sequence.

1. Press [M] for the Make-directory command. The Filer responds with this prompt:

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

2. You want to make a directory, so type [D]. The Filer responds with this prompt:

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

3. You enter this response:

```
#5:/WORKSTATIONS
```

Be sure to type this name in capital letters! If the root directory was protected with one or more passwords, the Filer would report: 'Error: invalid password' at this point. In such a case, you need to find out the required passwords from whoever initialized the SRM disc or installed the passwords. To create this directory, you need Write access rights in the root directory, and possibly Manager rights if they were specified.

For instance, if the password for Write access is PLEASE, you would specify:

```
#5:/.<PLEASE>/WORKSTATIONS
```

Alternatively, you might use the main volume password by specifying:

```
#5<VOL_PASS>:/WORKSTATIONS
```

The Filer should reply:

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

4. You answer [Y]. The directory is created, then the Filer announces:

```
Directory WORKSTATIONS made.
```

If the computers in the SRM configuration have Boot ROM 3.0 (or later version) which is able to boot from the SRM, you will also want to create a directory called SYSTEMS in the root. Repeat the steps just given, but instead specify that you want to create directory #5:/SYSTEMS.

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

5. Type [P] for the Prefix command. The Filer responds:

```
Prefix to what directory ?
```

6. Enter:

```
#5:/WORKSTATIONS
```

The Filer will respond:

```
Prefix is WORKSTATIONS:
```

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

1. Press [M]
2. `Make file or directory (F/D) ? D`
3. `Make what directory? SYSTEM`

4. `Directory is 'SYSTEM' correct? (Y/N) Y`

5. `Directory SYSTEM made`

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

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

Copying the System Files to SRM

You are now at the last stage! It is time to move the required files out into the new directories.

1. First prefix the current working directory to SYSTEM. Press [P] for the Prefix command.

2. Enter this directory specification:

   ```
   #5:/WORKSTATIONS/SYSTEM
   ```

   The Filer responds with this message:

   ```
   Prefix is SYSTEM:
   ```

3. Then insert the BOOT: disc in the drive you have been using and copy all the files on it into the new working directory. Press [F] for the Filecopy command. The Filer gives this prompt:

   ```
   Filecopy what file?
   ```

4. Specify that you want all files on the BOOT: disc to be copied by using the = wildcard as follows:

   ```
   BOOT:=,$
   ```

   The Filer will copy the files one after another.

Then repeat the above operation for each of the Pascal system discs (ACCESS:, SYSVOL:, etc). After this is done, the /WORKSTATION/SYSTEM directory contains the entire Pascal Workstation system.

Duplicating Links to System Files

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

1. Press [D].

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

2. You want to duplicate links rather than move links. Press [D]. The Filer will ask:

   ```
   Dup_link what file?
   ```

3. Answer:

   ```
   ?,#5:/WORKSTATIONS/SYSTEMnn/$
   ```

   Of course you should substitute a two-digit node number for nn each time (a leading 0 is required for single-digit node numbers). The "?" wildcard tells the Filer to ask if you want links each file in the source directory. Answer [Y] for every file except AUTOSTART and SYSTEM_P.

The Dup_link operation is very fast. It displays each file name as the links are made.

The last detail is optional. If any of the workstations in the SRM system have Boot ROM revisions 3.0 or later and will be expected to boot from the SRM instead of using local mass storage, you need to put a copy of the system Boot file in directory /SYSTEMS (not in /WORKSTATIONS/SYSTEM). The system Boot file (SYSTEM_P) is on the BOOT: disc shipped with the system; you probably have already made a copy of it in an earlier procedure. The Dup_link command can duplicate the file in a different directory.

1. Type [D] for the Duplicate link command.

   The Filer responds with this prompt:

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

   Respond with [D].

2. The Filer prompts with this question:

   ```
   Dup_link what file?
   ```

   Respond with:

   ```
   #5:/WORKSTATIONS/SYSTEM/SYSTEM_P,#5:/SYSTEMS/$ [Enter]
   ```

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

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

```
USERS
```

Then you can add subordinate directories like the following for each user:

```
USERS/TOM
USERS/DICK
USERS/HARRIET
```

SRM as the System Volume

At this point, you can make the /WORKSTATIONS/SYSTEMnn the system volume. You will first need to re-execute the TABLE program in order for unit #45: to be assigned to this directory. Press [X] at the Main Command Level, and enter this file specification:

```
/WORKSTATIONS/SYSTEMnn/TABLE.
```

Of course you will need to replace the nn with the node number of your workstation. Don't forget the period.

Now you can execute the Newsysvol command (at the Main Command Level) and specify **#45**: as the unit number. Then use the What command to verify that all of the subsystems (EDITOR, FILER, etc.) were found in the /WORKSTATIONS/SYSTEMnn directory. Changing the system volume will allow you to access the SRM copies of these subsystems by pressing keys such as [E] for Editor, and so forth.

Note

You should not prefix the working directory of unit #45: away from this directory.

Adding Modules to INITLIB

Now we will add modules DATA_COMM and SRM (on the CONFIG: disc) to INITLIB (on the BOOT: disc); actually, you will make a new INITLIB on the SRM that includes the drivers required for the SRM.

1. At the Main Command Level, press [L] to load the Librarian (note that the Librarian should be loaded from the SRM).
2. When you see the Librarian's prompt line at the top of the CRT, press [O] to specify the name of the (Output) file the Librarian will be creating.
3. Enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITNEW
   ```

4. Press [I] so you can specify an Input file, then enter:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITLIB.
   ```

   Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the input file.

5. Near the bottom of the CRT you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press [T] to transfer this module to the output file. After a few moments, the name of a new module (KBD) will appear. Each time a new module name appears, press T to move it to the output file. You should continue copying modules until the name LAST appears; **Don't copy the module LAST yet.**

6. Now you must get the required SRM drivers and include them in the Output file. First close the Input file by typing an [I] and then entering a null response.
7. Press [I] for an Input file and enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/DATA_COMM.
   ```

   Don't forget the period after the name.
8. When the module name DATA_COMM shows up near the bottom of the screen, press [A] which tells the Librarian to transfer all the modules in the file.
9. Then use the I command again to pick up the SRM input file, again being sure to type the period after the file name:

   ```
   #5:/WORKSTATIONS/SYSTEM/SRM.
   ```
10. Again transfer All by typing [A].
11. Enter an [I] (Input file) command with null response. This closes the SRM file.
12. Press [I] for Input and enter the file specification of the original INITLIB file:

    ```
    #5:/WORKSTATIONS/SYSTEM/INITLIB.
    ```
13. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing [M] for module and enter:

    ```
    LAST
    ```

    Then transfer it by typing [T].
14. You now have all the modules in your new library. "Keep" it by typing [K]. Then quit the Librarian by typing [Q].

Replacing INITLIB

Where you place the new version of INITLIB depends on which Boot ROM is in your machine.

- If you have Boot ROM 3.0 or later (but not 3.0L), then you will probably want to leave it in the /WORKSTATIONS/SYSTEM directory; it will be found there automatically when you boot from the SRM system.
- If you have an earlier Boot ROM or Boot ROM 3.0L, then you will need to replace the INITLIB on the BOOT: disc with the new INITLIB; this is required because these Boot ROMs cannot boot directly from SRM — they must use the BOOT: disc.

With Boot ROMs 3.0 and Later

1. Use the Filer's Change command to re-name the existing INITLIB (in /WORKSTATIONS/SYSTEM) to something like OLDINITLIB.
2. Use the Change command again to re-name the INITNEW file to INITLIB.
3. Re-boot your workstation to verify that the new INITLIB file works correctly.
4. Use the Filer's Dup-link command to link the new INITLIB to all /WORKSTATIONS/SYSTEMnn directories for the workstations that will be booting from the SRM. (You can alternately make custom INITLIB files for each workstation, if desired.)

With Earlier Boot ROMs

1. Press [F] to invoke the Filer.
2. Put in the spare copy of the BOOT: disc (*not* the original) into a drive. Press [R] for the Remove command. The computer responds with this prompt:

   ```
   Remove what file?
   ```

3. Answer:

   ```
   BOOT:INITLIB
   ```

   Note that there is no period after the file name this time.

4. Press [K] (Krunch) to pack all the remaining files on the disc to make the maximum amount of room for the new INITLIB. The Filer answers:

   ```
   Crunch what directory?
   ```

5. Answer:

   ```
   BOOT:
   ```

   Don't fail to type the colon after the volume name!

   The Filer will then say:

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

6. Answer [Y]. The computer then prompts:

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

Note

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

After the Krunch is complete, the filer prompts:

```
Crunch completed
```

7. Now when the Crunch is finished, you can Filecopy the INITNEW library file onto the new BOOT: disc. At the same time, you can re-name it INITLIB.

   Insert disc NEWLIB and press [F] for the Filecopy command.

   ```
   Filecopy what file?
   ```

   Answer:

   ```
   #5/WORKSTATIONS/SYSTEM/INITNEW.CODE,BOOT:INITLIB
   ```

   When the Filecopy finishes, you have a BOOT:INITLIB disc which contains the SRM drivers.

8. Verify that the new INITLIB works by re-booting your system.

Each Pascal workstation in the system with earlier (or 3.0L) Boot ROMs must boot using an INITLIB which has the SRM driver software installed. You may wish to make copies of the disc you've just created for each workstation. The disc can be copied using this Filer command sequence: [F] `#3,#3`. (You can alternately make custom INITLIB files for each workstation, if you want.)

Multi-Disc SRM

When an SRM system has more that one hard disc, you will need to modify, recompile, and execute the CTABLE program to allow access to these discs. This section describes how to perform this type of configuration change.

When more than one hard disc is installed on the SRM system, each disc must have a /WORKSTATIONS directory. If the directory is write-protected, then a /WORKSTATIONS/TEMP_FILES directory must be created. You may also wish to create another /SYSTEMS directory. Boot ROM 3.0 and later versions will search for bootable systems on each disc containing a /SYSTEMS directory.

CTABLE Modifications

Near the end of the CTABLE program, just above the manual `templates` section, a small section of code assigns Unit Table entries for the SRM. As shown below, the first `tea` entry provides a template for assigning unit #46: to the second hard disc connected to the SRM.

```
with SRM_dav do
  begin
    {  tea_srm( 46, sc, ba, du);  {free unless booting from HFS hard disc}
    tea_srm( 45, sc, ba, du);  {for possible use as the system unit}
  end; {with}
```

The *sc*, *ba*, and *du* fields are explained below:

sc Select code, such as 21.

ba Host node number, normally 0. It is the node number set on the card in the SRM console machine to which the Pascal system is connected.

du Disc volume number, normally 8 for the primary volume, or drive, and 9 for the second volume. Use the Volumes command at the SRM console to determine which value to use.

Just below the manual "templates" section of the CTABLE program is another section pertaining to units for the SRM.

```
{ prefix the primary and secondary SRM unit entries }

if not unit_prefix_successful('#5:/') then {do nothing};
  {tries to set up uvid for possible default unit assignment below}

{ if not unit_prefix_successful('#46:/?') then zap_assigned_unit(46);  {free}

{ NOTE: DO NOT UNCOMMENT THE ABOVE LINE IF YOU BOOT FROM AN HFS DISC! }

if not unit_prefix_successful('#45:'+srmsysprefix+srmnode(unitable^[45].sc)) then
  if not unit_prefix_successful('#45:'+srmsysprefix) then
    zap_assigned_unit(45);
```

If you remove the leading comment delimiter ({) from the #46: entry and remove the question mark from the literal `'#46/?'`, then Pascal will be able to recognize the second hard disc connected to the SRM.

If you wish to have a Unit Table entry for a particular directory path name, you can include the path name in the specification. For example:

```
if not unit_prefix_successful('#46:/USER/AL') then zap_assigned_unit(46);
```

If you make this modification be sure to activate its accompanying `tea_srm` procedure by removing the curly brace.

```
tea_srm( 46, sc, ba, du); {free}
```

With this modification, the system will boot with unit #46 assigned to the directory "/USER/AL" on the first SRM disc.

After all modifications have been made, you can compile CTABLE. Remember that you need to enable the `$search 'CONFIG:INTERFACE'$` Compiler option at the beginning of the program and make the INTERFACE library accessible at compile time. You will probably also want to link the resultant TABLE object file to itself with the Librarian to conserve disc space. See the procedures in the preceding section called Modifying the TABLE Program for explicit details. Also refer to *SRM Installation Manual*, chapter 4 for more examples.

Non-Disc Mass Storage 19

Introduction

Pascal 3.0 (and later versions) supports several types of "non-disc" mass storage:

- Internal memory (RAM)
- HP 98259A Magnetic Bubble Memory cards
- HP 98255A EPROM (Erasable Programmable Read-Only Memory) cards
- Cartridge Tape Drives (found in CS80-type disc drive units)

The Bubble and EPROM cards and tape drives provide non-volatile mass storage of programs and data; internal memory is volatile. All of them can be accessed through the File System. However, Pascal will not recognize either Bubble or EPROM cards until a few modifications are made to INITLIB and CTABLE (TABLE).

This chapter describes configuring and accessing Bubbles, EPROMs, and tapes. Using internal memory for mass storage is covered in the *Pascal User's Guide* and in the description of the Memvol command in the Main Command Level chapter.

Summary of Configuration Modifications

In order for the File System to recognize either the Bubble card or the EPROM card, you need to make the following configuration changes:

- Add the appropriate driver-module to INITLIB
- Modify the TABLE auto-configuration program. The source program (CTABLE) already contains the necessary templates; you only need to make a few simple changes to enable them.

Tape drives will be recognized without changes to INITLIB or the TABLE auto-configuration program.

Note

Shared Resource Manager (SRM) mass storage is discussed in Chapter 18 — "Special Configurations".

Mass Storage Comparison

The operating characteristics for various mass storage devices are compared in the following table.

Storage Devices

Characteristics	Flexible Discs	Bubble Cards	EPROM Cards	Memory Volumes	DC600 Tapes
Storage Capacity (bytes)	270 336 to 785 408	131 072 or 262 144	131 072[1]	variable[2]	16 000 000 or 67 000 000
Relative Access Speed	moderate	slow	fast	fast	slow
Read/Write Capability	yes	yes	no[3]	yes	yes
Usable as Boot Device	yes	yes[4]	yes[5]	no	no
Removable	yes	no	no	no	yes
Multiple Volumes	no	no	yes[6]	no	yes
Data Integrity	moderate	good	good	moderate[7]	good
Relative cost	low	high	moderate	moderate	low

[1] Size depends on EPROM device type. Sixteen 2764-type devices provide 131 072 bytes while sixteen 27128-type devices provide 262 144 bytes.
[2] Size is limited by available memory.
[3] EPROMs can be read just like RAM memory but must be programmed (written) with the HP 98253 EPROM Programmer Card.
[4] The CTABLE program must be modified to allow this boot device to be the default system volume.
[5] This device can be allowed as a boot device; however, there are several restrictions that apply. See the discussion of Booting from EDISCS for further information.
[6] Multiple volumes can be programmed into one EPROM Card, on 16 Kbyte boundaries.
[7] RAM memory reliability is dependent on power-source stability.

Using Bubble Cards

This section provides all of the information you will need to configure and access Bubble memory cards from the File System.

Power Constraints

Due to the amount of power consumed by a Bubble card when data is being transferred, no more than two Bubble cards can operate at the same time without exceeding the capacity of the power supply in the existing Series 200/300 Computers. It is further recommended that only one Bubble card be operating at the same time as any other "high-power" card (such as the HP 98620 DMA card).

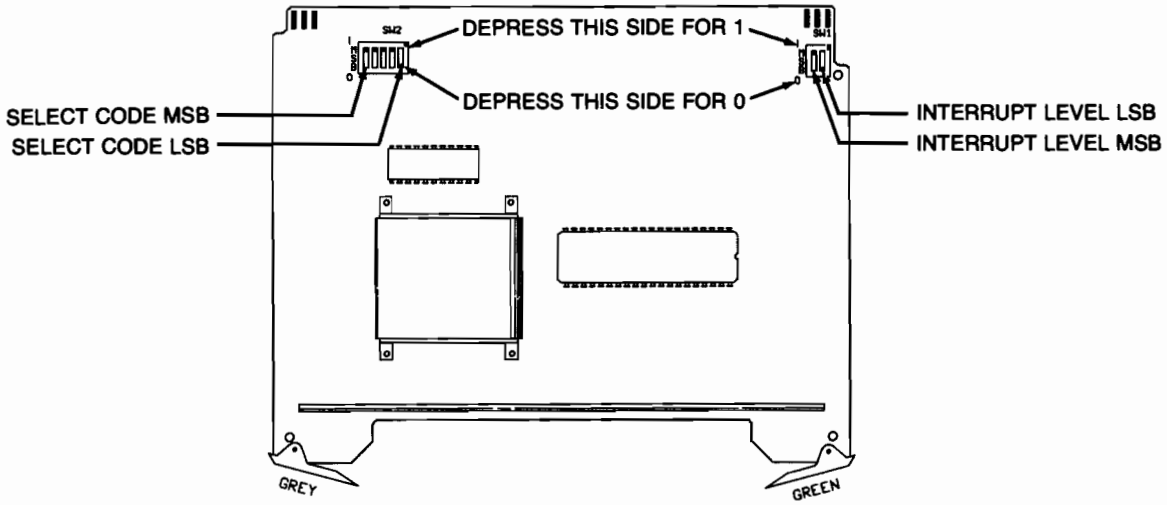
Bubble Card Configuration

If you have not already installed the Bubble card, see its installation note for complete details. Some of the installation information is repeated below for convenience.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

The Bubble card has two banks of switches. The large switch bank sets the select code while the small one controls the interrupt priority.



Bubble Card Switch Locator

Select Code

The Bubble card's select code is preset at the factory to select code 30. If this select code conflicts with any other interface present in the system, change it to some *unused* value from 8 through 31. Note the select code setting; it will be needed for the changes to the TABLE program.

Note

If you change the select code of the Bubble card from its factory default setting, you must also change the CTABLE program accordingly.

Select Code Switch Settings

MSB...LSB	Select Code		MSB...LSB	Select Code
01000	8		10100	20
01001	9		10101	21
01010	10		10110	22
01011	11		10111	23
01100	12		11000	24
01101	13		11001	25
01110	14		11010	26
01111	15		11011	27
10000	16		11100	28
10001	17		11101	29
10010	18		**11110**	**30**
10011	19		11111	31

Interrupt Priority

The interrupt priority switches have been preset to level 5. Each Bubble card should be set to a *unique* interrupt priority since the Bubble card may lose data if interrupts are not serviced quickly. This is especially true if you plan to make calls directly to the driver procedure or use the overlapped I/O capability.

Interrupt Priority Switch Settings

MSB...LSB	Level
0 0	3
0 1	4
1 0	**5**
1 1	6

If other interfaces have been installed which use interrupt level 5, change the switches on the Bubble card to the highest unused interrupt level in the range 3 through 6.

The Bubble card should now be ready to install in the computer. With the power turned *off*, install the card in the backplane. See the installation note if you have any difficulties.

INITLIB Driver Modules

The BUBBLE module is supplied on the LIB: disc (for double-sided disc configurations it is supplied on the ACCESS: disc). The IODECLARATIONS module recognizes Bubble cards as CARD_TYPE=8 (a field of the ISC_TABLE array in IODECLARATIONS).

Loading the BUBBLE Module

As with other driver modules, there are two ways to load the BUBBLE module:

- Execute it (with the Main Command Level eXecute command)
- Add it to INITLIB

Executing the module "permanently" loads the module, but must be done every time the system is booted. Adding the module to INITLIB eliminates having to load the module each time and allows the Bubble card to be a candidate for use as the system volume.

Adding BUBBLE to INITLIB

If you have two flexible disc drives or a hard disc (or SRM), the creation of the new INITLIB is relatively simple. If you only have a single flexible disc drive, you will need to create a memory volume large enough to hold the new library (about 200K bytes).

To create a memory volume, press [M] from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 400 unless you have added other modules to INITLIB; in that case more blocks are needed) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the Main Command Level chapter of Volume I of this manual.

Note

This procedure described here is correct for systems which have been supplied on single-sided discs. If your Pascal system was supplied on double-sided discs you should be aware that BUBBLE is on the ACCESS: disc, not the LIB: disc.

Note

If you are using a Series 300 computer you should replace the word "BOOT:" in the following section with "BOOT2:" unless your CRT is a 98546A.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the destination disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the destination disc.

   It is very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are now, no matter what happens to the new disc.

2. Invoke the Librarian. This is done by pressing [L] from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.

3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)

4. Now use the Librarian to create the new INITLIB.

   a. Press [I] and then enter the file specification by typing `#3:INITLIB.` and [Return] or [ENTER]. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

      When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press [F] to list all of the modules in INITLIB.

      The BUBBLE module can be inserted anywhere after the IODECLARATIONS module but *before* the module named LAST. (LAST must be the last module in INITLIB.)

b. Press [O] and enter `#4:BUBLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

   (This disc must *not* be removed until you have finished creating the new BUBLIB file.) If you are using a memory volume or other non-floppy mass storage, use the unit number of that volume.

c. Press [E] to enter the Edit mode. You should now see this prompt (in the middle of the screen):

```
F  First module: KERNEL
U  Until module: (end of file)
```

d. Press [U] and enter `LAST` as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing [C] once.

e. When the preceding copy is complete, press [A] to append a module to the BUBLIB Output file. The Librarian prompts with `Input file:`. Put the LIB: disc, or whichever disc now contains the BUBBLE module, in Unit #3 (*not* #4, which must not be removed). Enter this file specification: `#3:BUBBLE.`.

f. The Librarian now prompts with `Enter list of modules or = for all`. Enter =. After the BUBBLE module has been transferred to the BUBLIB library, the Librarian prompts with `Append done, <space> to continue`. If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the LIB: disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

   (If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message: `cannot open '#3:INITLIB', ioresult = 10`. In such case, don't worry. Remove the LIB: disc and insert the BOOT: disc, then press [I] and enter `#3:INITLIB.` as the Input file. Press [E] to return to Edit mode, and return to where you were previously by pressing [M] and entering `LAST` as the current module. Proceed with step g below.)

g. Press [T] to transfer module LAST to the BUBLIB file then press [S] to stop editing and [K] to keep the file.

h. You should now verify that the BUBBLE module was indeed copied to the Output file. Press [I] and enter `#4:BUBLIB.` as the new Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press [F] to get a File Directory listing.

i. If all modules are present, then press [Q] to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named BUBLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume or other mass storage, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy BUBLIB from the volume to the new disc, changing the file name to INITLIB in the process.

6. Re-boot the computer, which installs the new INITLIB containing the BUBBLE module.

Once the BUBBLE module has been installed, the Bubble card can be accessed by procedure calls. (The procedure calls will be discussed later.) To make the Bubble card available to the file system as a mass storage unit, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The CTABLE program, supplied on the ACCESS: disc (or CONFIG: disc), contains a "template" for the Bubble card. You can either use the Editor's Find command to find "templates" then "BUBBLE", or Jump to the end of the program and scroll up until you see the BUBBLE template shown below.

```
$if false$ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

Change `$if false$` to `$if true$`.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

This is the only modification that must be made to CTABLE for the system to recognize the Bubble card. It assigns unit number 42 to the Bubble card. If you are already using unit number 42, change the unit number to one that you are not using.

If you have more than one Bubble card or wish to have the Bubble card as a possible system volume, you should consider the following modifications.

Multiple Bubble Cards

If you install more than one Bubble card, a separate "tea" call must be made for each card. An example is shown below.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
    tea_BUBBLE(20,primary_dam,28);
    tea_BUBBLE(21,primary_dam,29);
    { tea_BUBBLE(3,primary_dam,30); {This would override #3}
$end$
```

For each tea_BUBBLE procedure call made, you should specify an unused unit table entry, the type of directory access method (the LIF DAM is recommended), and the select code (switch setting) of the Bubble card. Since these templates override the auto-configuration, the last entry in the above example would have overridden the device otherwise assigned to unit #3.

You can use the Filer's Volumes command to determine what units are being used.

Bubbles as the System Volume

The current TABLE program *already* contains code to support BUBBLE memory as a default system volume. This support is declared in the `{system unit auto-search declarations}` constants near the end of the "options" module. This constant tells TABLE to search through 7 possible system volumes. Note that unit number 42 (the default for the Bubble card) is included in the list. If you used a unit number other than 42 for the Bubble card, be sure to change the unit number in the search list above.

Compiling CTABLE

After all modifications have been made to the CTABLE program, the program must be compiled. If you do not know how to compile a Pascal program, see the Compiler chapter for details.

The resulting code file should be linked and stored as TABLE on the new BOOT: disc. Since CTABLE imports several operating system modules, you will need to make the ACCESS:INTERFACE (or CONFIG:INTERFACE) file accessible to the compiler (this file contains the interface text for the operating system modules). Note that for systems supplied on double-sided media, INTERFACE is found on the ACCESS: disc. To do so, you can either "uncomment" one of these compiler options (near the beginning of CTABLE.TEXT): `$search|#'CONFIG:INTERFACE'$` (or `$search|#'ACCESS:INTERFACE'$`) or add the INTERFACE file to the current System Library file. The linking procedure is described next.

Linking CTABLE

Once CTABLE.EXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of TABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled and resides in unit #3 as CTABLE.CODE. Since the linked version of CTABLE is usually less than 16K bytes, it will be put on the same disc that contains the original CTABLE.CODE file (probably CONFIG:) and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (Output) file directly onto the new BOOT: disc.

1. Press [L] to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press [I] and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press [H] to specify a new Header size; enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press [O] and enter `#3:TABLE.` as the Output file name. The trailing period will suppress the `.CODE` suffix.
5. Perform the actual linking.
   a. [L] — to start Linking. This will update the display.
   b. [D] — to toggle the DEFs (symbols) output to `NO`.
   c. [A] — to transfer All modules.
   d. [L] — to finish Linking.
   e. [K] — to Keep the Output file.
   f. [Q] — to Quit the Librarian.
6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did **not** include the BUBBLE module in the INITLIB, the File System will not recognize the Bubble card until you execute the BUBBLE module.
7. Re-boot the system using the new BOOT: disc. The Pascal Workstation system will now recognize the Bubble card.

Bubble Cards in the File System

After the BUBBLE module is installed and an appropriate TABLE program is executed, the Bubble card appears to the File System as a non-removable blocked device (mass storage volume). Any of the local mass storage directory access methods (DAMs) may be used, however the LIF DAM is recommended to allow use of the unit as a boot device and because of its low "overhead".

Executing the Filer's Volumes command will now show that a unit number has been assigned to the Bubble card. For example:

```
Volumes on line:
  1   CONSOLE:
  2   SYSTERM:
  3 # ACCESS:
  4 * SYSVOL:
  6   PRINTER:
 42 # VBUB:
Prefix is - ACCESS:
```

Unlike discs, Bubble memory units are initialized with the LIF DAM before being shipped. This means there is already a directory on the Bubble media. Use the Filer's List command to see the directory. For example, from the Main Command Level, press [F] to access the Filer, and then use the List directory command by pressing [L]. Specify #42 (or whatever unit number is assigned to Bubbles). Here is a typical display:

```
VBUB:              Directory type= LIF level 1
created 14-Apr-84 16.21.25 block size=256
 Storage order
...file name....     # blks     # bytes  last chng

FILES shown=0 allocated=0 unallocated=8
BLOCKS (256 bytes) used=0 unused=509 largest space=509
```

You can now use the Bubble card as you would any other LIF mass storage volume. It can be zeroed (all files removed) by the Filer's Zero command and it can be initialized by the MEDIAINIT program supplied with the system.

The Bubble Memory cards have access and timing characteristics similar to the Model 226/236 Computers' internal mini-floppy mass storage drives.

Your Bubble card should provide years of reliable, error-free operation. If you ever have cause to suspect the reliability of the Bubble card, make a back-up copy and and then try re-initializing the card before contacting your Sales and Service Office.

Error Correction

The Bubble Memory unit shipped to you has automatic error correction enabled. If some other memory unit (the hardware package containing the magnetic bubbles) is ever installed in the Bubble card, it should first be initialized by MEDIAINIT to ensure that automatic error correction is enabled.

The Bubble Device

The Bubble Memory unit installed in the Bubble card is a very stable non-volatile storage system. It is not easily damaged by external magnetic fields or mechanical abuse. It is, however, *strongly* recommended that the memory unit not be removed from the card. Removal of the memory unit from the card may damage the "boot loop" or the "seed bubbles".

The boot loop of a Bubble card is equivalent to the spared tracks record of a disc. If the boot loop is damaged, the memory will not function properly. A damaged boot loop may appear as permanent read/write errors, or more likely it will be detected by the TM (Transfer Method) when a UNITCLEAR operation is performed and reported as bad hardware. UNITCLEAR is performed on all units by the TABLE program and by a CLEAR I/O operation initiated from the keyboard (using the [Stop] key).

The memory of a Bubble unit is organized in tracks similar to a disc. Since a bit of information is indicated by the presence or absence of a bubble, information is written to a track by destroying or creating magnetic bubbles.

A magnetic bubble is created by splitting a seed bubble. If the bubble unit is removed or improperly installed a seed bubble may be destroyed or lost. This condition will appear as permanent read/write errors. If you suspect your bubble unit has been damaged in this way, contact your HP Sales and Service Office.

Initialization

The MEDIAINIT program on the ACCESS: disc is capable of initializing a BUBBLE device. The initialization process writes blanks to every location on the media, then writes a default directory to the unit. The only time MEDIAINIT should have to be used is when a Bubble Memory device not supplied by HP is placed on the card.

The Filer's Zero command can be used to remove all the files in the Bubble card. The procedure is similar to the Zero operation of discs. You can change the volume name and the number of directory entries but you should accept the default value for the size of the media.

If you do choose to initialize the Bubble card, execute the MEDIAINIT program and supply the appropriate unit number. Use the default value for all questions.

Interrupts and Overlapped I/O

Bubble devices require immediate interrupt service of relatively short duration. Since the Pascal Workstation File System performs *only* serial I/O, the problem of interrupt priority selection is reduced to ensuring that the BUBBLE module is placed in INITLIB after all other driver modules (but before module LAST). This will ensure that Bubble cards are checked before any other devices (on the same interrupt level) and therefore minimize the time required to service an interrupt.

When performing overlapped Bubble-card-to-Bubble-card transfers, best results are achieved when the destination priority is lower than, or the same as, the source priority. A priority configuration other than this will result in even poorer performance than if non-overlapped I/O is used because the two devices interfere with each other and cause several re-tries per transfer. This is not a problem on machines equipped with cache-memory hardware.

Using EPROM Memory

This section introduces you to the programming and operating characteristics of the HP 98255 EPROM card and the HP 98253 Programmer card. With these cards and Pascal 3.0 (or later versions), you can copy files and volumes into EPROMs.

Overview

EPROMs are high-speed memory devices used for storing programs or other information. The HP 98255 EPROM card and the HP 98253 Programming card support 2764, 27128, and equivalent types of EPROMs.

The EPROM devices are not supplied with the EPROM card. You will have to purchase them separately through an electronic-supply vendor or other source. You probably will also need to purchase an ultra-violet (UV) light source to erase the EPROMs.

The storage capacity of an EPROM can be determined by the final digits of the part number. For example, a 2764-type device contains 64 Kbits (65 536 bits) while a 27128-type device contains 128 Kbits (131 072 bits). Up to 16 EPROMs can be placed on one card; this means that one card provides 131 072 bytes of storage using the 2764-type EPROMs or 262 144 bytes using 27128-type EPROMs.

The data in an EPROM can be read just like RAM memory, however, a special process is needed to program (write) the data into EPROMs. The HP 98253 Programmer card is used for this purpose. An EPROM is programmed by applying a short "burn" pulse while the data being programmed is applied to the output pins. The timing and control of this operation is handled by the Programmer card. Once the EPROMs have been programmed, the Programmer card is no longer needed in the system and can be removed. (With the power turned-off of course!)

An EPROM can be erased (all bits set to "1") by exposing it to a high level of ultra-violet (UV) light. Once erased, the EPROMs can be reprogrammed with new data. Check the EPROM manufacturer's specifications for details on the type of UV light source needed and the recommended exposure time.

Configuration Changes Required

There are two changes you need to make to the "standard" configuration in order to use EPROMs:

- Add module(s) to INITLIB.
- Modify the TABLE source program (CTABLE.TEXT)

You may also need to set switches on the cards and install EPROM devices.

INITLIB Driver Modules

In addition to supporting the operations of the HP 98255 EPROM card and the HP 98253 Programmer card, Pascal 3.0 (and later versions) supports the use of EPROMs as a mass storage volume. Transferring a volume into EPROMs would create what could be called an "Eprom-DISC" or "EDISC".

The support modules include:

- The EPROMS module is included on the LIB: disc (ACCESS: disc for double-sided system discs). The module may be installed by either executing it or by using the Librarian to include it in INITLIB.
- The EDRIVER module, also included on the LIB: disc (ACCESS: disc for double-sided system discs), provides *read/write* capability for performing various operations with an EPROM and Programmer card pair. The EDRIVER module can be "P-loaded" or linked to an application program when read/write capability is needed.
- The EPROM Transfer Utility (ETU.CODE) included on the LIB: disc (ACCESS: disc for double-sided system discs) allows mass storage volumes to be transferred to EPROMs. When environmental conditions limit the reliability of floppy discs, or when it is desirable to have quick access to commonly used programs or data, a copy of a mass storage volume can be transferred to EPROMs. Transferring a volume to EPROMs creates an "EDISC".

  ETU can also be used to transfer DATA, ASC, UX and TEXT files to EPROMs. This capability allows arbitrary bit-patterns to be transferred to EPROMs.
- The CTABLE.TEXT file on the CONFIG: disc contains a "template" section to assign unit numbers to EDISCs.
- The Pascal IODECLARATIONS module recognizes a Programmer card as CARD_TYPE = 9 (a field of the ISC_TABLE array). This same version also recognizes Bubble memory cards.

You do not have to load any modules before using the ETU program since it already has the necessary drivers included in its code. When you are finished programming the EPROMs, if you transferred a whole volume to the EDISC, the EPROM module should be added to INITLIB to provide file system access to EPROMs. This process is described later in this chapter.

Programmer Card Installation

If you have not already installed the Programmer card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

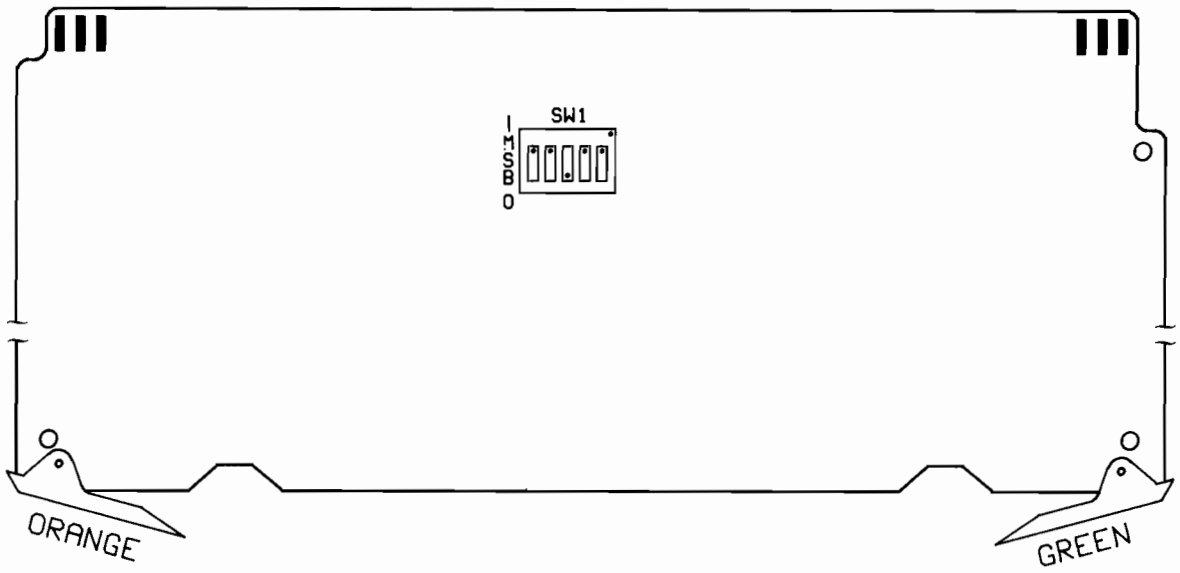
The purpose of the HP 98253 Programmer card is to program (write) information into the EPROMs on the HP 98255 EPROM card. Once the information has been programmed, the Programmer card can be removed from the computer's backplane.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

Perform the following steps to install the Programmer card:

1. Check the select code switch on the Programmer card. The HP 98253 Programmer card's select code has been preset to 27 at the factory. If this conflicts with any other I/O card in the system then change it to an unused select code. If more than one Programmer card is installed, set each card to a unique select code.



Programmer Card Switch Location

Select Code Switch Settings

MSB...LSB	Select Code		MSB...LSB	Select Code
01000	8		10100	20
01001	9		10101	21
01010	10		10110	22
01011	11		10111	23
01100	12		11000	24
01101	13		11001	25
01110	14		11010	26
01111	15		**11011**	**27**
10000	16		11100	28
10001	17		11101	29
10010	18		11110	30
10011	19		11111	31

2. With the computer power turned *off*, install the Programmer card in the computer's backplane. The Programmer card's ribbon cable will be connected to an EPROM card later.

When more than one Programmer card or EPROM card is installed at the same time, the ribbon cable can be connected to different EPROM cards without turning off system power. Be sure that no read or write operation is taking place when the cable is exchanged.

CAUTION

THE PROGRAMMER CARD'S CABLE MUST **NOT** BE REMOVED OR CONNECTED WHEN THE EPROM CARD IS IN USE.

A small light-emitting diode (LED) on the Programmer card indicates when system power is on. (It does **not** indicate when the card is in use.)

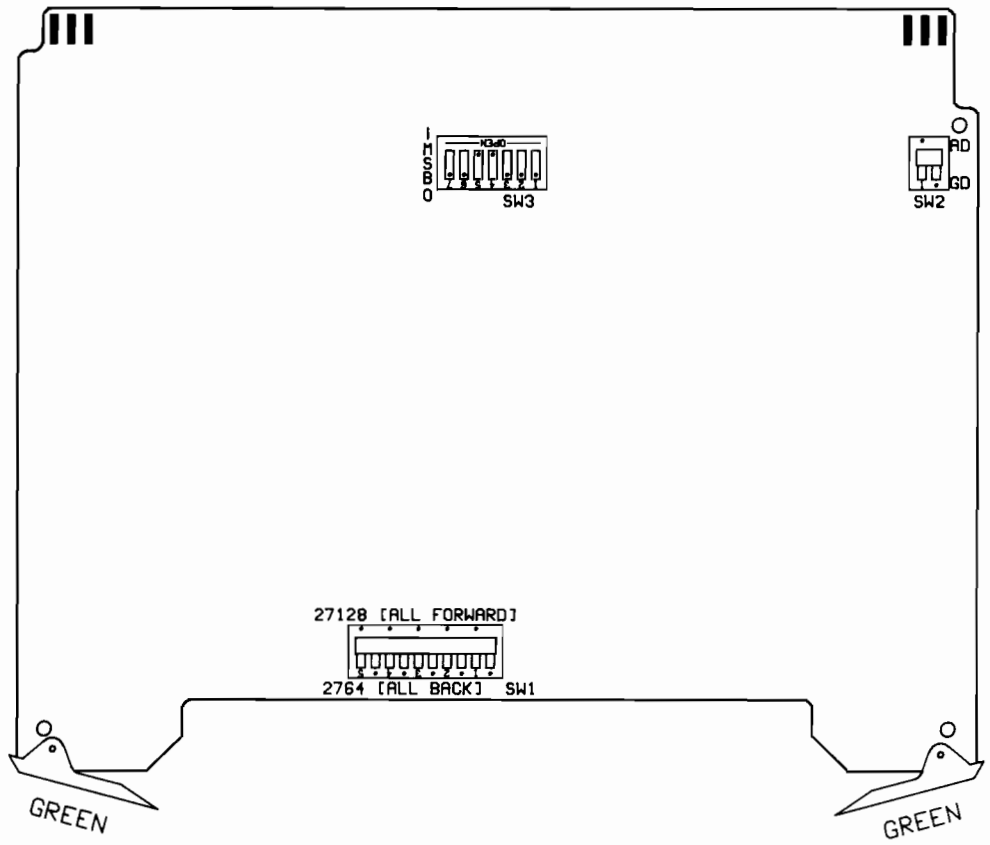
EPROM Card Installation

If you have not already installed the EPROM card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

Before installing an EPROM card in the computer's backplane, you need to check and set the card switches. There are three sets of switched on the card.

- EPROM-type switch (SW1)
- Address-response switch (SW2)
- Address switch (SW3)

The position of these switches is shown in the following illustration:



EPROM Card Switch Locations

The largest switch is the "EPROM-type" switch. It tells the card's hardware what capacity of EPROM to expect. All segments of this switch are "ganged" together to configure all 16 sockets for either 2764-type or 27128-type EPROMs. You cannot mix two different types of EPROMs on one card, but you do not need to completely fill all 16 sockets with EPROMs. If you only partially fill the card, use pairs of EPROMs (upper and lower byte socket-pairs) and fill the lowest numbered sockets first.

The smallest switch on the card is the "DTACK" switch and it controls the card's response when it is addressed (i.e. whether it should respond like ROM or RAM memory). This switch, which can be set for AD (Automatic DTACK) or GD (Generate DTACK), must be set to AD for Series 200 and GD for Series 300 for the EPROM card to appear in the computer's ROM memory space.

Note

The modules provided with Pascal 3.0 (and later versions) only support EPROM cards which are addressed in the ROM address space. Set the "DTACK" switch to AD for Series 200 and GD for Series 300.

The third switch determines the base memory address of the card. Special care must be taken to ensure that the address space selected does not overlap another EPROM card or ROM card. The EPROMs on the card are "memory mapped" (in pairs) by ascending socket number. For example, byte 0 is the first location in socket 0U, byte 1 is the first location in socket 0L, byte 2 is the second location in socket 0U, byte 3 is the second location in socket 0L, and so on.

Note

If you have a ROM-based language system, do **not** set the EPROM card's switches to the same address space used by the ROM Language. For instance, the ROM version of the HPL Language System is addressed at $10 0000 ("$" indicates a hex address) and extends up to $12 0000. The ROM 1.0 version of the BASIC Language is addressed at $2 0000 and extends to $2 4000, while the ROM 2.x versions begin at $8 0000 and vary in size.

To see where these ROM-based systems reside, you can check for the presence of "ROM headers" (contents $F0FF) which are located on 16 Kbyte boundaries, beginning at $2 0000 and extending through the Auto-DTACK range of addresses. Auto-DTACK extends to $20 0000 for cache-memory processor boards (i.e., machines with "U" suffix such as the HP 9836U) and $40 0000 for non-cache-memory processor boards (such as the Model 236A).

Although the switches can be set to make the EPROM card appear almost anywhere in the computer's address space, the following table shows the recommended settings. When the smaller capacity EPROMs are used, multiple cards can be addressed 128 Kbytes apart; cards filled with the larger capacity devices must be addressed 256 Kbytes apart.

Address Switch Settings

Switch Settings	EPROM Card's Base Address for Programming		
MSB LSB	Hex Start Address	Decimal Address (2764-type devices)	Decimal Address (27128-type devices)
0 0 0 0 0 0 1	$2 0000	131 072	
0 0 0 0 0 1 0	$4 0000	262 144	262 144
0 0 0 0 0 1 1	$6 0000	393 216	
0 0 0 0 1 0 0	$8 0000	524 288	524 288
0 0 0 0 1 0 1	$A 0000	655 360	
0 0 0 0 1 1 0	$C 0000	786 432	786 432
0 0 0 0 1 1 1	$E 0000	917 504	
0 0 0 1 0 0 0	$10 0000	1 048 576	1 048 576
0 0 0 1 0 0 1	$12 0000	1 179 648	
0 0 0 1 0 1 0	$14 0000	1 310 720	1 310 720
0 0 0 1 0 1 1	$16 0000	1 441 792	
0 0 0 1 1 0 0	$18 0000	1 572 864	1 572 864
0 0 0 1 1 0 1	$1A 0000	1 703 936	
0 0 0 1 1 1 0	$1C 0000	1 835 008	1 835 008
0 0 0 1 1 1 1	$1E 0000	1 966 080	
0 0 1 0 0 0 0	$20 0000	2 097 152	2 097 152
0 0 1 0 0 0 1	$22 0000	2 228 224	
0 0 1 0 0 1 0	$24 0000	2 359 296	2 359 296
0 0 1 0 0 1 1	$26 0000	2 490 368	
0 0 1 0 1 0 0	$28 0000	2 621 440	2 621 440
0 0 1 0 1 0 1	$2A 0000	2 752 512	
0 0 1 0 1 1 0	$2C 0000	2 883 584	2 883 584
0 0 1 0 1 1 1	$2E 0000	3 014 656	
0 0 1 1 0 0 0	$30 0000	3 145 728	3 145 728
0 0 1 1 0 0 1	$32 0000	3 276 800	
0 0 1 1 0 1 0	$34 0000	3 407 872	3 407 872
0 0 1 1 0 1 1	$36 0000	3 538 944	
0 0 1 1 1 0 0	$38 0000	3 670 016	3 670 016
0 0 1 1 1 0 1	$3A 0000	3 801 088	
0 0 1 1 1 1 0	$3C 0000	3 932 160	3 932 160
0 0 1 1 1 1 1	$3E 0000	4 063 232	

Once the EPROM card's switches have been set, install the EPROM devices on the HP 98255 EPROM card. Be very careful when installing the EPROMs on the card, since the pins are easily bent. Both the EPROMs and the sockets have notches to indicate the proper orientation. See the installation manual for details.

With the power switched *off*, install the EPROM card in the computer's backplane.

Multiple EPROM Cards

If more than one blank EPROM card is installed in the computer's backplane at the same time, be sure each EPROM card is addressed to different memory locations. The lowest addressed card should be programmed first. Blank EPROM cards can not be detected by the Pascal system unless they are connected to the Programmer card.

Cable Connections

When you have finished installing the Programmer and EPROM cards, you can connect the ribbon cable from the Programmer card to the desired EPROM card. The cable connection defines and establishes the "card-pair" for programming operations.

The Programming Utility

The ETU program supplied with Pascal 3.0 supports the following operations for an EPROM and Programmer card-pair:

- Display current Programmer and EPROM card information
- Check for blank space on the EPROM card
- Transfer a mass storage volume to EPROM (EDISC creation)
- Transfer DATA, TEXT, UX, or ASC files to EPROM (user-defined patterns)

The exact action taken in a transfer operation depends on the type of file involved. All transfers are done in two passes through the data. The two passes perform the same actions except that the data is actually programmed (written) only during the second pass.

Note

All file types other than TEXT, UX and ASC are treated as DATA files.

Transferring Volumes to EPROM

When you specify that a volume is to be transferred to EPROM, the ETU program assumes that an EDISC is to be created. The EDISC will appear to the File System as a mass storage volume as a floppy disc, but with much faster access. The maximum size of the volume depends on the capacity of the EPROMs installed in the card. The largest EDISC that can be created contains 256K bytes, since EDISCs can not cross EPROM card boundaries.

Once an EDISC has been created, you should not copy the EDISC to any other mass storage volume, as it contains extra data that only EPROMS can handle correctly. Individual files may be copied.

Booting from EDISC

Boot ROM 3.0 and later versions can boot from an EDISC. Booting from these devices is like booting from any other mass storage media; the system is copied into RAM and executed from there rather than from the EDISC (unlike ROM-based systems which execute directly from ROM).

EDISC as the System Volume

Pascal can also recognize an EDISC volume as the system volume; however, since the system volume is used by the system to store all temporary files, I/O error 18 ("Device is write-protected") will be reported whenever the system attempts to write to this "write-protected" device.

AUTOSTART and other normal stream files will not work if the system volume is an EDISC. (Normally, when a file is Streamed, the file is copied to the file named STREAM on the current system volume; this is not possible with EDISCS, since they are effectively "write-protected.") You should use the AUTOKEYS file to perform autostart functions from these devices. Other stream file names must contain a [*] specifier which indicates that the stream-file prompt feature is disabled. See the description of the Stream command in the Overview chapter for further details of using prompts in Stream files.

EDISC Headers

When a volume is transferred to EPROMs, an EDISC "header" is first generated and programmed into the EPROMs. The Boot ROM can detect the header and make that information available to the file system. In other words, if a volume is transferred to EPROMs, special information is added that allows the Boot ROM to detect and possibly boot from the EDISC.

Boot ROM 3.0 and later versions check for EDISC headers (and other types of headers) on 16 Kbyte boundaries, starting at 128 Kbytes ($2 0000) and continuing through the Auto-DTACK range of addresses; this range extends to $20 0000 for machines with cache-memory processor boards (i.e., computers with "U" suffix such as the Model 236U), and $40 0000 for machines without cache-memory processor boards (such as the HP 9836A). This searching operation effectively divides the address space into 16 Kbyte "blocks".

Since the Boot ROM will check for an EDISC header on every 16 Kbyte block boundary, more than one EDISC can be programmed onto a single EPROM card. There are 8 blocks (numbered 0..7) on an EPROM card using the small capacity EPROMs and 16 blocks (numbered 0..15) using the large capacity EPROMs.

To prevent the Boot ROM from accidentally interpreting the contents of a block boundary as an EDISC header, the utility program writes binary zeros (hexadecimal pattern $0000) into the boundary locations searched by the Boot ROM. The volume's data is appropriately mapped around the block boundaries. The mapping operation is completely handled by the system, but this does mean the EDISC volume will require a few bytes more than the original volume.

The total number of bytes needed to program a volume can be computed by taking the source volume's size and adding 18 bytes for the EDISC header and 2 bytes for each 16 Kbyte boundary crossed. If the last sector of the volume is unused, the extra bytes can be truncated without loss of data, as long as the volume is not HFS.

Transferring Files to EPROM

The ETU program can be used to transfer DATA, TEXT, UX, or ASC type files to EPROMs. If the file type is **not** TEXT or ASC, the files will be programmed into EPROMs so as to create an exact bit for bit copy. If the file type is TEXT or ASC then *only the data parts* are programmed into EPROM (not the data separators and other "overhead"; this is equivalent to reading a line from the file into a string with a READLN statement and then "burning" the contents of the string.)

Unlike volumes, individual files transferred to EPROMs without the "directory" information of a volume cannot be detected by the file system. If you write a program to access a file that was programmed into EPROMs, you will have to tell your program where to find it. Even if only one file is to be transferred to EPROMs, you might consider putting the file in a volume and transferring the volume if you want file system access to it.

Not only do you have to keep track of the location of individual files transferred to EPROM, you must be sure that the data does not accidentally appear to the Boot ROM as a "ROM header". The Boot ROM searches for a two-byte header pattern (F0FF hexadecimal) at 16K byte intervals in the computer's ROM space.

The header pattern is not likely to occur in TEXT or ASCII files, however, a DATA file programmed into EPROMs may contain such a bit-pattern, and if the pattern occurs on a 16 Kbyte "block" boundary, unpredictable results may occur at boot time. The ETU program does not check for this condition.

Preparing a Transfer

Before starting the utility to transfer a volume or file to EPROMs, you must decide what you want to transfer. There are some restrictions that may influence your decision.

- The total number of bytes transferred must be less than the total capacity of the EPROMs. Excess bytes will be truncated. It is unlikely that a truncated file will be very useful.
- If the "source" volume is larger than the current available space on the EPROM card, the volume will be truncated. Since LIF volumes contain all of their directory information at the beginning of the volume, you can truncate the unused sectors at the end of a volume with relative impunity. This is not the case for HFS volumes.

For the purposes of this discussion, it has been decided to transfer the Pascal Editor and Filer to EPROMs. This will allow fast access to the programs without requiring as much RAM memory as is necessary to "P-load" both of them. The number of bytes required for both the Editor and Filer is less than 120K bytes so both programs will easily fit on the EPROM card even if the smaller capacity EPROMs are installed.

Note that programs are **not** usually executed in EPROMs; rather, a copy of the program is made in RAM memory and then the copy is executed. When you quit the program, and the copy is no longer needed, the RAM memory used for the copy is free to be used by other programs. This has advantages over a program that is "P-loaded" since a "P-loaded" program remains in RAM memory until the next boot operation.

The volume containing the programs to be transferred to EPROMs should not be any larger than necessary since the entire volume will be transferred, including any unused sectors. Once the volume has been transferred to EPROMs, there is no way to go back and fill the unused sectors in the volume. Therefore, for our example, the best approach will be to create a memory volume just large enough to hold both the Editor and Filer. This will be the volume that is transferred to EPROMs.

Creating a Memory Volume

A memory volume with up to eight (8) files needs two (2) "system" sectors, plus one (1) sector for directory information, plus enough sectors to hold the files. The size of the Filer is about 250 sectors (64000 bytes), and the size of the Editor is about 234 sectors (59904 bytes). Thus, in our example, we need 3 + 1 + 484 sectors, or a total of 488 sectors.

The Memvol command "thinks" in 512-byte blocks not in 256-byte sectors. Therefore, to create the correct size memory volume, we need an even number of sectors (round-up). The total number of blocks is then 484/2 or 242 blocks.

These calculations are for LIF memory volumes, HFS memory volume size calculations are difficult to ascertain as there is a varying amount of overhead to be accounted for depending on the size of the memory volume created. To give you an idea though, here are three examples of HFS memory volumes and the overhead required.

1. A memory volume created with 200 blocks gives a LIF volume with an available space of 101632 bytes. When HFS is installed on the volume (with minfree=0 [see HFS chapter for explanation]), the useable space drops to 43008 bytes, the overhead for HFS being 59392 bytes.
2. A memory volume created with 400 blocks gives a LIF volume with an available space of 204032 bytes. When HFS is installed on the volume, the useable space drops to 129024 bytes, the overhead for HFS being 75776 bytes.
3. A memory volume created with 800 blocks gives a LIF volume with an available space of 408832 bytes. When HFS is installed on the volume, the useable space drops to 292864 bytes, the overhead for HFS being 116736 bytes.

It is therefore probably only worthwhile creating *larger* HFS memory volumes, on which the overhead is relatively low. Note that the use of HFS on EPROMS is not recommended nor supported.

From Pascal's Main Command Level, press [M] to create a memory volume. Answer 242 to the "Number of Blocks" question, and answer 8 to the "Number of Directory Entries" question.

When you have created the memory volume, Filecopy the Editor and Filer from the ACCESS: disc. Then use the Filer to Change the volume name from RAM: to ESYS: (the volume name will also be transferred to EPROMs).

The "Empty sockets" command of the transfer utility can be used to protect EPROMs from being programmed if there are a large number of unused sectors in the volume being transferred to EPROMs. The ETU program will then detect that the volume being transferred is larger than the available space and allow you to truncate the unused bytes. Be sure that it does not truncate part of a file!

The EPROM Transfer Utility

The EPROM Transfer Utility program (ETU.CODE) is included on the LIB: disc (for double-sided system discs it supplied on the ACCESS: disc). This utility provides a convenient method of programming the EPROMs on a HP 98255 card. Either single files or entire volumes can be transferred to EPROMs with this utility.

If you haven't already executed ETU.CODE, do so now. When the utility is executed, it automatically searches for the Programmer card connected to an EPROM card. If a card is missing or incorrectly installed, you will get one of the following messages.

```
*** NO PROGRAMMER CARD IN SYSTEM ***

NO EPROM CARD ATTACHED TO PROGRAMMER CARD
```

If the system does not recognize the Programmer card, turn power off and check the select code switch settings. You should check that each switch segment is toggled correctly and that no other interface card is set to the same select code.

When the Programmer and EPROM cards have been correctly installed and connected to each other by the Programmer card's cable, the main menu is displayed. (Note: the space-bar was pressed to remove the release date and copyright notice from the following display.)

```
ETU: Transfer Configure Blankcheck Quit ?

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL
```

There are four functions available from the main menu: Transfer, Configure, Blank check, and Quit. Each of these functions will be explained on the following pages.

Your display may differ depending on the select code setting of the Programmer card and the capacity of EPROMs installed in the EPROM card. If more than one Programmer card is installed in the system, all operations will use the "active" Programmer card. If more than one EPROM card is installed in the system, all ETU operations will affect only the EPROM card connected to the (active) Programmer card's cable.

The various functions are activated by typing the first letter of the appropriate operation (for example, [C] for Configure). Lettercase does not matter. Incorrect letters are ignored except if the program is under stream control. When streaming, incorrect letters will abort the program and the stream file.

All operations can be aborted by typing [Shift]-[Select] ([SHIFT]-[EXECUTE]) for single character answers or [Shift]-[Select] and then [Return] or [Enter] ([SHIFT]-[EXECUTE] and then [ENTER]) for multi-character answers.

In stream file operations, answers to optional questions are automatic and are the *first* option given in the prompt. For example:

- For a YES/NO question ending with "(Y/N) ?" — The stream answer is "Y"
- For an ABORT/TRUNCATE question ending with "(A/T) ?" — The stream answer is "A"

Configuration

From the main menu, press [C] to display the configuration sub-menu. The main menu is replaced with a sub-menu which lets you change the select code, the burn rate, and specify any empty EPROM sockets.

```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ?

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL
```

The configuration functions are explained next. Pressing [Q] will return you to the main menu.

Select Code

When only one Programmer card is in the system, it is automatically chosen as the active Programmer card and the select code is properly set.

If you have more than one Programmer card installed in the system and wish to change operations to a different Programmer card, press [S] for Select code. The following prompt will appear at the bottom of the display:

```
New select code (27) ?
```

The number in parentheses indicates the select code of the currently selected Programmer card. You may either press [Return] or [ENTER] to accept the current select code or type the select code of a different Programmer card. If the select code you type is valid, the display will be updated with the new information. An error message will be displayed if the new select code does not correspond to a Programmer card.

Burn Rate

Pressing [B] will cause the Burn rate to change from SLOW to FAST or from FAST to SLOW (the display is automatically updated).

Note

All EPROMs may be programmed at the slow burn rate. Some EPROMs are not guaranteed to retain the pattern if the faster rate is used. Check the EPROM manufacturer's specifications before using the faster programming rate.

If the FAST burn rate is specified and a location fails to accept the data, the burn rate will automatically be switched to SLOW.

The FAST burn rate programs at 13.1 ms/byte while the SLOW burn rate programs at 52.3 ms/byte. The card circuitry can program both upper (even address) and lower (odd address) bytes in parallel so the effective rate is 13.1 or 52.3 ms/word. Therefore, programming every location in a full set of large capacity EPROMs using the FAST burn rate will take about an hour.

Note that the Burn rate is a global attribute not associated with a particular Programmer or EPROM card.

Empty Sockets

An empty EPROM socket appears to be an erased (blank) EPROM. This condition can not be detected until an attempt is made to program a pattern into such a location.

Pressing [E] allows you to specify which sockets of an EPROM card do not contain EPROMS; the information is used in the calculation of the capacity of an EPROM card. EPROMs must be used in pairs and up to 8 pairs of EPROMs may be used in one card.

```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ? E

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL

SOCKET (PAIR) NUMBER ?
```

For example, if you answered "7" to this question, the display would be updated as follows:

```
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7 empty
```

In this manner, you can specify all of the empty sockets. If you make a mistake, re-execute the command with the same socket number; the program will again mark the socket pair as usable.

An error message is displayed if the socket pair number is out of range.

Quitting the Sub-Menu

Quitting the Configure sub-menu will return you to the main menu. Once you have completed the configuration for the active card-pair, the next step is to check for available space in the EPROMs.

Blank Check

Pressing [B] from the main menu will show the used and unused space (according to how many EPROMs you've told the program are on the EPROM card connected to the active Programmer card). A blank EPROM has all of its bits set to binary 1's. Thus, a blank byte would contain the hexadecimal pattern FF.

Unused space will be shown at the bottom of the display. For example:

```
ETU: Transfer Configure Blankcheck Quit ? B

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL

BLANK CHECK
            0 - 131071 (131072)
```

The number in parenthesis indicates the size of the unused space (in bytes). The two numbers separated by a hyphen indicate the relative address of the unused space within the active card.

The above display indicates that the entire EPROM card is unused. Bytes 0 through 131 071 are blank and the total number of contiguous blank bytes is 131 072.

If no blank bytes can be found on the current EPROM card, the following error message will be displayed:

```
NO BLANK SPACE FOUND
```

Typically, after an EPROM has been programmed, there will be some bytes containing the hexadecimal pattern: FF. These bytes will appear to the program as "blank" and the Blank Check option will list them as follows:

```
address - address (size in bytes)
address - address (size in bytes)
```

The above lines are repeated as many times as required, in groups of 6, with a prompt to press the spacebar to continue between each group. The addresses given are relative to the base address of the EPROM card. The size is likely to be only a few bytes for addresses that actually contain data. (A hexadecimal FF programmed into EPROM looks like a "blank" location.) The last entry is likely to indicate any truly "blank" space. The sockets you've specified as empty are not counted.

Now that the available space has been determined, you are ready to transfer a file or a volume to EPROMs.

Transfer

Pressing [T] from the main menu will prompt you for information about the transfer operation. ETU makes some assumptions to try to help you.

The display will show the following:

```
ETU: Transfer Configure Blankcheck Quit ? T

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL       4UL
1UL       5UL
2UL       6UL
3UL       7UL

TRANSFER OPERATION
Source (ESYS:) ?
```

The ETU program assumes that a volume will be transferred to EPROMs. The volume name in parenthesis is the current prefixed volume. You may accept the volume name by pressing [Return] or you may type another volume name. If you specify both a volume name and a file name then ETU assumes that a single file is to be transferred to EPROMs. If you do specify a different volume or a file, the display will be updated accordingly.

When transferring a volume to an EPROM card, if no "blank" block is found on the EPROM card the following message is given:

```
*** NO BLANK BLOCK ON THIS EPROM CARD ***
```

The program will then display:

```
Start at eprom block offset (0) ?
```

The value in parenthesis indicates the lowest numbered "blank" block. (If every block has been programmed, a zero is displayed.)

Or if a file was specified:

```
Start at eprom byte offset (0) ?
```

For a file, the value in parenthesis is always zero.

If the default value in parenthesis is acceptable, press [Return] to begin the transfer operation. Optionally, you may specify a different block offset or byte offset. See the previous sections on Transferring Volumes and Files for the details about offsets.

If there is insufficient space on the EPROM card for the transfer, the ETU program will prompt:

```
DATA EXCEEDS EPROM SPACE BY xxxx BYTES
Abort transfer or Truncate file (A/T) ?
```

Where xxxx represents the number of excess bytes.

A reply of [A] or [Shift]-[Select] ([SHIFT]-[EXECUTE]) will cancel the operation. A reply of [T] will cause the transfer of only as much data as will fit on the EPROM card. If this happens during the execution of a stream file, the transfer operation will abort and the stream file will be terminated.

A transfer is a two-pass operation. The first pass checks the data and the EPROMs. The second pass actually programs the data into the EPROMs and verifies that it has been stored correctly.

Unless an error occurs, the transfer is automatic from here on.

Check Failure

Check failure is detected during the first pass. The byte to be programmed is matched against the byte on the EPROM card. If the EPROM can not be made to contain the new pattern then a CHECK FAIL results. (An EPROM's "0" bits can not be changed to "1" bits.)

```
CHECK FAIL AT ABSOLUTE ADDRESS aaa
BYTE POSITION bbb FROM START LOCATION
EPROM SOCKET un BYTE nn
```

Where "aaa" is the absolute machine address of the byte which will not program or did not program. The position "bbb" is the byte index (from 0) of the byte in the file. The position is also identified by EPROM ("un" is socket identifier: for example, U1 or L4) and "nn" is the byte offset (from 0) within the identified EPROM.

Burn Failure

If an EPROM fails to accept a byte of data using the FAST burn rate, the utility automatically switches to the SLOW burn rate, updates the display, and attempts to continue.

If the burn rate is already SLOW when a byte fails to program properly, then a "BURN FAIL" occurs. The utility is aborted and a message is displayed. For example:

```
BURN FAIL AT ABSOLUTE ADDRESS 3997696
BYTE POSITION 65536 FROM START LOCATION
EPROM SOCKET 4U BYTE 0
```

If the programming fails exactly on a socket boundary ("BYTE 0" in the example above) check to see if the socket is empty or if the EPROM is improperly installed (bent pins).

Quitting ETU

Pressing [Q] from the main menu will quit the utility and exit to the Pascal Main Command Level.

This concludes the operations of the ETU program. Once the EPROMs in an EPROM card have been programmed, the Programmer card can be removed from the system. (With the power switched off of course!)

Before the File System can recognize an EDISC, the EPROMS Transfer Method (TM) module must be loaded into the system and a modified version of the CTABLE program must be compiled and executed. (The ETU program has its own driver module and could locate the EPROM card since it was connected to the Programmer card.)

Loading the EPROMS Module

The EPROMS module is supplied on the LIB: disc (or ACCESS: disc for double-sided system discs). As with other driver modules, there are two ways to load the module:

- Execute it (with the eXecute command at the Main Level) — you will need to run TABLE after doing this.
- Add it to INITLIB and re-boot

Executing the module "permanently" loads the module, but must be performed every time the system is booted. Adding the module to INITLIB eliminates having to load the module and run TABLE each time you re-boot the system.

Adding the EPROMS Module to INITLIB

If you have two disc drives, the creation of the new INITLIB is relatively simple. If you only have one disc drive, you will need to create a memory volume large enough to hold the new library (about 200K bytes).

To create a memory volume, press [M] from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 400) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the Main Command Level chapter of Volume I of this manual.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the disc.

Note

If you are using a Series 300 computer you should replace the word "BOOT:" in this section with "BOOT2:" unless you are using a 98546 display card as the primary display.

(It is a very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are, no matter what happens to the new disc.)

2. Invoke the Librarian. This is done by pressing [L] from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.
3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)
4. Now use the Librarian to create the new INITLIB.
    a. Press [I] and type `#3:INITLIB.` and [Return] or [ENTER] to enter the Input file. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

       When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press [F] to list all of the modules in INITLIB.

       The EPROMS module can be inserted anywhere after the IODECLARATIONS module but before the module named LAST (it must also precede module BUBBLE, if that module is present). In this example, the module will be included as the next-to-last module in the new INITLIB.

    b. Press [O] and enter `#4:EPLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

       (This disc must *not* be removed until you have finished creating the new EPLIB file.) If you are using a memory volume, use the unit number of the memory volume.

    c. Press [E] to enter the Edit mode. You should now see this prompt (in the middle of the screen):

```
F  First module: KERNEL
U  Until module: (end of file)
```

    d. Press [U] enter `LAST` as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing [C].

    e. When the preceding transfer is complete, press [A] to append a module to the EPLIB Output file. The Librarian prompts with `Input file:`. Put the LIB: (or ACCESS: for double-sided system discs) disc, or whichever disc now contains the EPROMS module, in Unit #3 (*not* #4, which must not be removed). Enter `#3:EPROMS.` as the Input file specification.

    f. The Librarian now prompts with `Enter list of modules or = for all.` Enter `=` to specify all modules. After the EPROMS module has been transferred to the EPLIB library, the Librarian prompts with `Append done, <space> to continue.` If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the CONFIG: (or ACCESS:) disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

       (If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message: `cannot open '#3:INITLIB', ioresult = 10`. In such case, don't worry. Remove the CONFIG: disc and insert the BOOT: disc, then press [I] and enter `#3:INITLIB.` as the Input file. Press [E] to return to Edit mode, and go back to where you were previously by pressing [M] and entering `LAST` as the current module. Proceed with step g below.)

g. Press [T] to transfer module LAST to the EPLIB file. Then press [S] to stop editing and [K] to keep the file.

h. You should now verify that the EPROMS module was indeed copied to the Output file. Press [I] and enter `#4:EPLIB.` as the Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press [F] to get a File Directory listing.

i. If all modules are present, then press [Q] to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named EPLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy EPLIB from the memory volume to the new disc, changing the file name to INITLIB in the process.

6. Re-boot the computer, which installs the new INITLIB containing the EPROMS module.

To make the EPROM card(s) available to the File System as mass storage units, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The Pascal CTABLE program contains a "template" for EPROM cards. You can either use the Editor's Find command to search through all occurrences of the `EPROM` token until you find the template, or Jump to the end of the program and scroll up until you see the EPROM template shown below.

```
$if false$ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

To activate the template, change `$if false$` to `$if true$` as shown in the following example:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

The template assigns the lowest addressed EDISC to Unit 42. It should be noted that this unit number is also the default for Bubble cards (and for the secondary cartridge tape driver) and may have to be changed to some other unit number more appropriate to your peripheral configuration.

EDISCs are recognized according to their relative addresses in the ROM address space of the system. The EDISC with the lowest address is assigned sequence number 0, the second lowest is assigned sequence number 1, and so on.

If you have more than one EDISC, your template might appear as follows:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
  tea_EPROM(27,primary_dam,{ sequence number } 1);
  tea_EPROM(28,primary_dam,{ sequence number } 2);
  tea_EPROM(31,primary_dam,{ sequence number } 3);
$end$
```

To force recognition of an EDISC (or multiple EDISCs), call the procedure TEA_EPROM with the appropriate unit number, DAM identifier, and sequence number.

The connection between unit number and address is made when a CLEARUNIT call is made to the TM. This implies that if the address switches of the EPROM cards are changed, the cards may be assigned different Unit Table entries.

In the Unit Table, the SC field is -1 and the sequence number is stored in the DV field.

Compiling CTABLE

Once the necessary modifications have been made to the CTABLE program, the program should be compiled. Since CTABLE imports several operating system modules, you will need to make the CONFIG:INTERFACE file accessible to the compiler (ACCESS:INTERFACE for systems supplied on double-sided media). This file contains the interface text for the operating system modules. To do so, you can either "uncomment" one of the following compiler option (near the beginning of CTABLE.TEXT):

```
$search 'CONFIG:INTERFACE'$
```

or

```
$search 'ACCESS:INTERFACE'$
```

or add the CONFIG:INTERFACE file to the current System Library file. Remember that for systems supplied on double-sided discs, INTERFACE is on the ACCESS: disc, not the CONFIG: disc. The linking procedure is described next.

Linking CTABLE

Once CTABLE.TEXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of CTABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled as CTABLE.CODE on unit #3. Since the linked version of CTABLE is usually less than 16K bytes, it will be put on the same disc that contains the CTABLE.CODE file and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (output) file directly onto the new BOOT: disc.

1. Press [L] to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press [I] and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press [H] to specify a new header size. Enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press [O] and enter `#3:TABLE.` as the Output file. The trailing period will suppress the `.CODE` suffix.
5. Perform the actual linking.
    a. [L] — to start Linking. This will update the display.

b. [D] — to toggle the DEFs (symbols) output to NO.

c. [A] — to transfer all modules.

d. [L] — to finish linking.

e. [K] — to keep the output file.

f. [Q] — to quit the Librarian.

6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did *not* include the EPROMS module in the INITLIB, the Pascal file system will not recognize the EPROM card until you install the EPROMS module.

7. Re-boot the system using the new BOOT: disc. The File System will now recognize the EPROM card.

EPROM Cards in the File System

After the necessary modifications have been made, and the system re-booted, you can use the Filer's Volumes command to see an EDISC.

For example:

```
Volumes on line:
  1   CONSOLE:
  2   SYSTERM:
  3 # ACCESS:
  4 * SYSVOL:
  6   PRINTER:
 42 # ESYS:
Prefix is - ACCESS:
```

Use the Filer's List command to see the directory. For example:

```
ESYS:              Directory type= LIF level 1
created  7-Jan-87  11.34.20 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

EDITOR                 228      58368  7-Jan-87
FILER                  224      57344  7-Jan-87
FILES shown=2 allocated=2 unallocated=6
BLOCKS (256 bytes) used=452 unused=1 largest space=1
```

You may now use EPROMs almost as you would any other write-protected mass storage volume. Remember, an EDISC should not be copied to another mass storage volume.

This concludes the EPROM installation and programming information.

Using Cartridge Tapes

This section describes use of the Streaming Cartridge Tape Drives, such as the HP 9144, for mass storage operations. If you have one of the Command Set '80 Series Disc Drives, you may also have a tape cartridge drive integrated into the machine for backup.

Tape Drives Supported

The currently supported Cartridge Tape and Disc/Tape Drives include the following HP products: HP 9144, HP 7908, HP 7911, HP 7912, HP 7914, HP 7942, and HP 7946.

Tape Lengths

There are two lengths of DC600 tapes: 150 feet and 600 feet; these tapes have capacities of 17 and 67 Mbytes, respectively. Both tapes can be directly accessed by the Pascal File System.

Tape Access Methods

The Pascal system provides three methods of computer-controlled tape access. The first is a utility program with capabilities similar to the integrated disc/tape product's "switch" backup. This is called TAPEBKUP and is described in Chapter 20 — Backup Utilities. The Operating and Installation Manual that came with the product may describe a method of off-line "switch" backup, involving the use of *save* and *restore* switches located on the tape drive itself. While these switches do provide full-volume image backup capability, they are intended for service-personnel usage only.

The second is a backup utility supplied with Pascal 3.2 and later versions which allows incremental and selective backup and restore of files. This utility, called BACKUP, is also described in Chapter 20. It is the most powerful backup method and produces tapes that are compatible among BASIC, HP-UX, and Pascal workstations.

The third method is "direct" access to the tape with the Pascal File System, a method which can be used for selective backup of files and logical volumes, even those not on a CS80 disc.

If you wish to access the tape as a file system, see Chapter 20 – "Using the File System for Direct Tape Access".

CAUTION

The cartridge tape drives are intended for use as streaming devices. Thus, using these tapes for direct access and selective back-up, although supported, may cause accelerated wear or damage to the tape drive and tape. In other words, use these tapes only for limited back-up and emergency purposes, not for normal file system calls in user programs or as part of a boot sequence.

Backup Utilities 20

Introduction

Pascal 3.2 has two backup utilities which, depending on the one chosen, enable you to make a master copy of the data on your disc or discs and retrieve this data at a later date so that it may be read by the same or a different operating system. Also, this chapter discusses the topic of accessing a cartridge tape as a file system.

The Backup Utility

The Backup Utility (`BACKUP.CODE`) is a program that enables you to copy some or all files on a disc onto a backup medium, such as a tape or flexible disc. This utility differs from Tape Backup (`TAPEBKUP.CODE`) in that it can recognize the structure and individual files on a disc, whereas the Tape Backup Utility performs a simple bit-for-bit copy of the whole disc.

The format in which the data is stored is such that it cannot simply be used as an additional copy of your working disc. The files are written sequentially in *cpio(4)*[1] format irrespective of whether the source file system was WS1.0, LIF, HFS or SRM. It is an archive.

An **archive** can be one or more cartridge tapes (or flexible discs) onto which files have been copied in *cpio* format. This is a subset of the *cpio* command of HP-UX which also allows a file to be the destination of a *cpio* operation. On the Pascal Workstation, the **archive medium** is a tape or flexible disc (or one of a set of tapes or discs).

What is Saved on the Backup?

In addition to saving the files, the backup contains the volume names and BASIC MSUS (Mass Storage Unit Specifier), the complete pathnames (on HFS and SRM backups), and the HFS ownership and permissions. SRM passwords are not saved. A backup can span more than one media (tape or disc) but the user must remember the proper sequence.

The Tape Backup Utility

The Tape Backup Utility (`TAPEBKUP.CODE`) is a program that enables you to copy the complete image of a disc onto a tape, or vice-versa. It also provides operations for certifying tapes and verifying the readability of either discs or tapes.

It is important to note that the utility only provides a complete image backup, and does not provide a selective file or volume backup. So, the backup must be restored to the same or larger capacity disc drive. A limited amount of selective backup is possible by using the Pascal file system for "direct" access to the tape.

How do I know which utility to use?

Unless you want to perform a complete image backup of your disc using a tape drive which uses the same controller as the disc, it is almost certain that you should use the Backup Utility.

[1] *cpio* is a supported command in HP-UX. A description of the format is found in the HP-UX Reference (Section 4 — File Formats). The Pascal Workstation and BASIC use the same format as the *cpio* command invoked with the `-c` option.

Using the Backup Utility

Note

A tape must be certified before it can be used with the BACKUP utility. HP supplied tapes are already certified before delivery. Other tapes can be certified with the TAPEBKUP utility.

Purpose

The purpose of this utility is to allow you to make a backup of your file(s) and have them accessible to a BASIC, Pascal or HP-UX user. It is possible to back up files from WS1.0, LIF, HFS, and SRM discs onto a single archive due to the flexibility of the *cpio* format used.

Either cartridge tapes or flexible discs may be used as archive media. However, once a tape has been used as an archive medium, it can only be used as an archive medium (by either BACKUP or TAPEBKUP) until it is re-initialized (using `MEDIAINIT.CODE`). After being re-initialized, it can be used for other file system operations (for example, Zeroing it with the Filer). It is not necessary to re-certify the tape during this MEDIAINIT.

How to Invoke the Utility

The Backup Utility is supplied on the HFS: (or HFS3:) disc. The utility file name is `BACKUP.CODE` and it can be eXecuted from the Main Command Level. Upon execution, this program will give you the following options:

- **Full Backup** — Backup of all specified files that exist in your file system. This can include files on more than one volume.
- **Incremental** — A backup of all files which were created or modified after a specified date.
- **Restore** — Restore of your backup data onto your working storage volume(s).
- **Table of Contents** — A display of the table of contents of an archive.
- **Quit** — Leave the utility.

These options appear in the form of a menu:

```
BACKUP:  Full  Incr.  Restore  Table-cont.  Quit?
                [Version 3.2]
     Copyright 1987 Hewlett-Packard Company
            All rights are reserved.
```

Multi-media Operation

For larger systems using high capacity discs (for example the 7937 disc, which has over 500M-bytes of storage space), you will probably make a multi-media archive. Since only one tape drive can be used, you will have to insert and remove the media yourself.

If your backup procedure requires that you use more than one tape (or disc) to hold all the files from your mass storage volumes, you should keep the following points in mind.

1. **ALWAYS** number your archive media **as you use them**. After you have made the backup may already be too late!
2. When necessary, the utility will automatically prompt you to insert a new medium, both for making a backup and when restoring your files onto your working disc.
3. If you failed to follow the advice given above and find that your media are out of order, during reading of the archive, BACKUP will recognize something is wrong. During a Table-of-Contents operation, an automatic resynchronization will be done, and some files will not be listed. Restore asks whether you wish to resynchronize in case of error; the default answer is No. If you choose not to resynchronize, Restore will abort if it finds an error. If you choose to resynchronize, you may be able to recover most files, if nothing is seriously wrong. If the media are in the wrong order, or if some of the data is unreadable, there is a possibility that the following files will appear on the wrong volume, possibly overwriting files that have the same name. This can only happen if the archive is a multi-volume backup. There will be some loss of data. This facility is also designed so that the loss of data is a minimum if one of your tapes is damaged or is not of the right format (`cpio -c` format).
4. All archive media must have been initialized before starting to create a Full or Incremental archive. If an uninitialized medium is encountered, the BACKUP program will abort.

Procedure for a Full Backup

On invoking the Full Backup by typing [F], the following prompt will appear:

```
Destination medium for backup ?
```

The unit number of the drive containing your archive media should be entered now, e.g. #41 for a tape drive. It is possible to redirect the logging information which normally appears on the screen, to a file or the printer. This is done by entering an optional second parameter specifying the file where this information should go. Note that the information will always be printed to the screen, in addition to this "logfile". Typical responses would be:

```
  #41: [Return]
or
  #3:,listfile [Return]
or
  #41,PRINTER: [Return]
```

The system will then check that the specified volume is on-line before responding. If the backup medium is a flexible disc that has more than one unit table entry, the system will give an appropriate warning that the contents of other units may be destroyed. For example, when the backup medium is a flexible disc drive which contains a disc of HFS format the unit number to enter would be #43, to which the response by the system would be:

```
Device: HP9153 removable disc, 705, 0, 0
Logical unit #43 - 'hfs3:'
WARNING: this will also destroy:
 #3  V3:

Checkread the destination (Y/N, default is N)?
```

The checkread is a bit-for-bit comparison between the buffer read from the source and that written to the destination. If there is any discrepancy found between these two sets of data, the BACKUP utility will abort. Irrespective of the answer given for this prompt, the final prompt before proceeding with the backup operation is:

```
Are you SURE you want to proceed ? (Y/N)
```

If you press [Y], you will be prompted for the names of the files to back up:

```
Source directory or file to backup ?
```

The response to this prompt can be a volume number (#11:), a file name (optionally including volume name and directory path, as well as SRM passwords), a set of files defined using wildcards (e.g. `=.TEXT`), or, by using the wildcard "=" without a filename, all files **and all files in any subdirectories encountered** on the volume or directory specified. Combinations may be used; i.e. if the above example, `=.TEXT`, is entered, this will back up all files in the current directory of the default volume which end in ".TEXT" and all files ending in ".TEXT" found in all subdirectories.

After saving all of the specified files, the prompt will be repeated until a null string is entered by pressing only the [Return] key. Saving all of the specified files may take quite a while, but the name of each file backed up is listed on the screen just after it is saved.

A full backup may require more than one medium to hold all of the files. If more are needed, you will be prompted:

```
Output medium full, insert medium 2 in #43.
When volume is ready, press <return> to continue.
```

The utility will then check the new medium and prompt you to confirm whether or not to proceed. With this utility it is possible to use as many media as are required to do a backup. In order to be able to restore all files correctly, you must mark the sequence of the media. The utility does not know the sequence of the archive media and will get "confused" if given media in the wrong order during a restore or table-of-contents operation.

Procedure for an Incremental Backup

On invoking the Incremental Backup by entering [I], the system will prompt you with the following:

```
Incremental backup date ?
```

All files which have changed or been created on or after the date entered will be copied to the archive. Files unchanged since this date will not be copied. The format for entering the date is the same as for the Version command specified in the "Main Command Level" chapter of Volume I of this manual (for example, `15-Feb-87`). There is no default date and simply pressing [Return] will bring you back to the BACKUP main menu.

The procedure from this point on is exactly the same as for the Full Backup.

Restore Procedure

The Restore option is selected by pressing `R`. This will cause the utility to give the following prompt:

```
Source medium for restore ?
```

Your response in this case should be the unit number of the drive containing the archive medium, for example #3. As with the Backup option, you can redirect the logging information to a file or printer by using the same input format as described previously. The system will continue, providing it has received a legal response, with:

```
Restore unconditionally (Y/N, default is N) ?
```

During a restore, unless `Y` is entered for the above prompt, the files which will be restored from the archive are:

- Files with unique file names or pathnames, i.e. files which have no counterpart on the volume to which they are being restored.
- Files with names which are not unique but which are newer than the files with the same name on the volume to which they are being restored.

Occasionally there will be a need to restore all the files on the archive, irrespective of whether or not the files on the archive are newer than the corresponding existing files on the file system. This can be achieved by entering `Y` for the above prompt, which will initiate an unconditional restore of all the files on the archive.

During a Restore you may wish to rename the files to different volumes. This allows you to restore the files to a different unit without changing their pathnames.

```
Rename destination volume (Y/N, default Y) ?
```

Responding with a `Y`, or simply pressing [Return], gives you the opportunity of redirecting files on the archive to different volumes and/or directories than the ones they originally came from.

Note

Responding with a `N` means that you must have the same volumes on-line as when the backup was made.

Next you are asked if you wish Restore to resynchronize if it encounters unreadable or incorrect data on the archive. See the previous discussion on the possible dangers of resynchronizing under the heading: "Multi-media Operation" before choosing an answer.

Finally, the utility will prompt you to enter the filenames of the files to be restored:

```
Source files to restore (default is all) ?
```

Simply pressing [Return] will cause all files on the archive to be restored. It is permitted to use wildcards here, for example entering `=.TEXT` will instruct the utility to restore all files ending with ".TEXT" regardless of the rest of the name or the path. The default to copy all files is only valid the first time the prompt appears.

```
Source files to restore ?
```

Note that the default is now no longer defined. This prompt will re-appear until you give a null entry (by pressing [Return]), which will signify that you wish to start the restore. It is therefore possible to copy all files ending in ".TEXT", ".CODE" or ".ASC" in one operation by answering the above prompt four times with:

```
=.TEXT [Return]
=.CODE [Return]
=.ASC [Return]
[Return]
```

In a similar manner, wildcards can be used to specify that only particular directories are to be restored. If HFS was associated with the volume `hfs11:` when the backup was made,

```
hfs11:USERS/TESTCODE/=
```

specifies that only the files and directories in `/USERS/TESTCODE` are to be restored.

Having exited from filename entry mode, you will be prompted:

```
Are you SURE you want to proceed? (Y/N)
```

Assuming that the input here is [Y], the restore will begin; if you have chosen to redirect files, you will be required to answer the following prompt a number of times depending on the number of times you responded to the "Source Directory or File to back up ?" prompt while making the archive.

```
Destination volume/directory to restore to (default is volume name)?
```

Backup's resynchronizing capability is active (by default) during a Table of Contents operation and can be activated during Restore by answering the prompt:

```
Resynchronize (Y/N, default N) ?
```

This capability is particularly useful when performing a restore involving a sequence of media when one of the media is found to be damaged or unusable. It permits you to start the restore with a medium which is not the first in the sequence in which the archive was made. Some data will not be restored but, providing the sequence is adhered to once resynchronizing has been completed, the amount will be the minimum possible. If the system finds an error with one of the media, during a Restore with resynchronize active, it will report it in the following way:

```
Out of phase; resynchronizing.
```

Assuming that the operation was successful, a confirmation will be given:

```
Resynchronized after skipping 99999 bytes.
```

and a warning at the end of the Restore option will give a reminder:

```
Unable to restore at least 999 files.
```

Caution should be exercised when restoring files which were previously stored in a hierarchical file system onto a WS1.0 or LIF directory, as there is a possibility of duplicate filenames occurring. For example confusion will occur when different files which have the same name, but were stored in different directories of an HFS disc, are restored onto a LIF or WS1.0 disc. If unconditional restore has been selected, and more than one file to be restored has the same name after removing its directory path, only the last one encountered on the tape is restored.

Given the same conditions, should you choose conditional restore (by answering [N] to the prompt), then:

- If a file with the same name exists in the LIF or WS1.0 directory, only the first file with a later date and time on the archive can be restored. If no files on the archive are newer, the existing file will remain.
- If no file with the same name exists in the LIF or WS1.0 directory, only the first file with that name on the archive will be restored.

Should there be *n* files with the same name on the archive, and you wish to restore the *n*th one to a LIF or WS1.0 disc, find its complete name (e.g. `MYDISC:/users/me/myfile`) and restore it by fill directory path and filename, redirecting to the LIF or WS1.0 disc. For the above example, you would restore unconditionally, redirecting say to #3, and specify "`/users/me/myfile`" for the filename.

Procedure to obtain a Table of Contents

To invoke this option, press `T` when the Backup menu is displayed. This option allows you to list a group of, or all files in a specified volume or directory. You will be prompted:

```
List content of what medium ?
```

You should enter the unit number of the device where the archive medium is located. The information on the archive is not stored in a volume that can be examined by the Filer or other utility programs, so the only correct method of accessing the medium is by using the unit number **not** a volume name. In addition, you may specify a listing file to receive the names of the files so that the filenames can be checked more easily. For example:

```
#41:,PRINTER:   or   #3:,#5:/PRINTER/LISTFILE.ASC
```

The option will continue by prompting:

```
List what files (default is all) ?
```

This prompt is similar to the "`Source files to restore (default is all) ?`" prompt in the Restore option. By pressing [Return] only, all the files on the medium will be listed. As described in the Restore option, it is also possible to selectively list files by using wild-cards, i.e. all ".TEXT" files or all files starting with the characters "Tech". This prompt only appears once. To list files on a different medium, or to list different types of files on the same medium, you must run the program again, selecting Table-of-Contents again.

Note

If you re-direct the output to a file, and this file is created and updated on the medium which you are trying to list, an error could occur.

Quit

This returns you to the Main Command Level.

Limitations of the Utility

The BACKUP utility is a comprehensive tool to make backups of your files and restore them but, like any utility, its capabilities are limited. To help you get the most out of the BACKUP utility you should be aware of the following:

- Archives made of SRM file systems do not include any information about passwords which may be associated with certain files. When these files and directories are restored, all password protections must be redone manually.
- An SRM file that cannot be read without a password cannot be copied to the archive without specifying the password. See the SRM Administrator for the SRM volume password if these "unreadable" files must be backed up.
- SRM files which are linked to other files will not be restored as linked files.
- If you make a backup of an HFS disc and restore these files onto an HFS disc, the access information is also restored along with the last modification date.
- HFS files which were linked at backup time will have their links restored *providing* the files are restored onto an HFS disc.
- There is no direct equivalent to the *Verify* option in TAPEBKUP provided with this utility. The CHECKREAD option (during full or incremental backup) will cause BACKUP to compare each written block to the original data. The CHECKREAD operation doubles the amount of time to perform a Full or Incremental backup.
- Using this utility, it is possible to make a backup of **all** files on an HFS disc even if there are files for which a Pascal or BASIC user has no read or write permission.
- There are certain types of "special" files that HP-UX creates which the Pascal Workstation cannot handle. These files include device special files, pipes, etc. A Filer Extended listing will show these special files as type `Hp-ux` but their `t-codes` will all be zero. Generally, files of this type cannot be created or opened by the Pascal Workstation. However, BACKUP can save these files in the archive. To restore these files, the HP-UX command `cpio -icx` must be used.

Using the Tape Backup Utility

Concepts and Terminology

Single and Dual Controllers: With the CS80 integrated disc/tape products, the standard option is for the disc and tape drives to share a common controller. The disc is unit 0; the tape is unit 1. One of the features of the shared-controller product is its ability to transfer data directly from unit 0 to unit 1 or vice-versa, without having the data travel through the host computer. This utility was written specifically to support this mode of operation, as it is the most efficient method for complete backup.

There is an option for the integrated disc/tape products where the disc and tape drives each have their own dedicated controller. Each controller has a separate HP-IB port and bus address, and no logical association with the other one. As a result, the "switch" backup capability is not available with this option. Likewise, *this utility does not support the dual controller option.*

Source and Destination Mis-matches: To be consistent with the product's built-in "switch" backup capability, the utility's Medium-copy operation allows all combinations of source and destination sizes, even those which might seem illogical. Thus, if you have more than one of the disc drives in this family, be sure to mark the type of drive which is backed up on each tape. For instance, if you were to restore a tape backup of a 7908 onto a 7911, much of the 7911 would be inaccessible until you re-Zero'ed it appropriately.

Tape Certification: Tape certification is a procedure very similar to hard disc initialization. Even though the tape comes pre-formatted from the manufacturer, it needs thorough testing, with a possible sparing of bad blocks, before it is ready for use. The addresses of spared blocks are entered into a sparing table and those blocks are never used again. While the tape certification process is somewhat lengthy, tapes usually need to be certified only once during their lifetime. Tapes purchased from Hewlett-Packard are already certified.

Tape Auto-sparing: Any time problems occur in the reading of a tape block, the tape controller will record this fact on the tape's permanent log, and then automatically spare out the troublesome block during the next write operation to it. This way, the tape actually tends to get better with usage; slightly marginal blocks that may have escaped detection during certification can be spared later. Note, however, that if a tape is re-certified, the previous sparing information is lost, and all defective blocks will have to be re-discovered.

The utility may in certain instances print "Tape certification in progress", and then almost immediately print "Tape certification completed". In this case, the tape was determined to already be certified, so it was *not* re-certified; it merely went through an optimization of its sparing tables.

Tape Unload Sequence: A loaded tape must go through a logical unload sequence before the tape drive will allow you to physically eject it. A tape unload sequence can be *initiated* either by the front panel *UNLOAD* switch or by the utility. Either way, the tape will then go busy for some period of time, to position it for unloading and to update its permanent logs. A minute or two later, you may hear a buzzing noise made by the tape drive heads as the sequence completes and the busy light extinguishes. You may now physically eject the tape.

When the utility prints "Tape unload request completed", it means that the request to the tape drive to *initiate* the unload sequence has completed. You will have to wait for the unload sequence itself to complete before you will be able to eject the tape.

Verification: Verification is a read without the transmission of data back to the host. The device still does its internal data integrity checks, although it usually inhibits the automatic retry mechanism employed by normal reads. In a verify, the data is not actually compared to anything; the device merely verifies that it can read the data correctly.

To Verify or Not Verify a Tape: Explicit verification of a tape takes as much time to do as normal reads or writes. Thus, in deciding whether to verify or not, you must weigh the time it takes to do the verify versus the extra assurance provided by it. With the present series of integrated disc/tape drives (i.e.,7908, 7911, 7912, and 7914), tape verification *is* recommended.

With the stand-alone HP 9144 Tape Drive, however, the drive incorporates a special read-after-write head, which allows verification of the readability of the data *as it is being written.* With these drives, explicit verification is *not* recommended, although it can still be performed.

If a Disc Doesn't Verify: If a disc gives trouble while verifying, the recommended procedure is to save its contents to a tape if desired, then re-initialize it using MEDIAINIT.CODE found on the ACCESS: disc. MEDIAINIT will perform a two-pass error rate test on the entire disc, and then intensively test further any blocks with which the disc controller "remembers" having had trouble. All bad blocks will be spared. After MEDIAINIT completes, the saved contents of the disc can be restored from the tape if need be.

In performing a save of the contents of the troublesome disc mentioned above, the utility may report bad blocks on the source, although not necessarily, since a verify inhibits read retries while a copy does not. In such a case, a best guess of the bad blocks' data would be sent to the tape, and the copy operation would complete. The tape would now contain one or more blocks with corrupted data, but it would "verify" correctly, assuming that it and the tape drive were good. Likewise, after restoring the data back to the freshly-initialized disc, the disc would have the tape's identically corrupted data, and it too would now "verify" correctly.

If a Tape Doesn't Verify: If a tape gives trouble verifying, *do not re-certify it*; merely repeat the write operation to the tape again. The utility always uses the tape in auto-sparing mode.

Specifics on 7914 Backup: To be consistent and fully compatible with the 7914's "switch" backup behavior, the utility

- Always requests two tapes
- Doesn't complain if they are not long tapes
- Doesn't complain if the two tapes do not correspond to each other

Even though rigorous checking is not provided, if you exercise moderate caution you shouldn't have any problems.

With a save, the utility always writes the "first half" tape first, followed by the "second half" tape. With a restore, the utility allows you to insert the tapes in either order; an internal "copy start address" field on the tape specifies which area to the disc to restore it to. The utility also prints out the source and destination start addresses for each copy segment, so that you can detect it if you accidentally restore two "first half" tapes or two "second half" tapes.

How to Invoke the Utility: The utility is quite simple to use. Its user interface is similar to the other Pascal subsystems. The `TAPEBKUP.CODE` utility is delivered on the ACCESS: disc. Like any other program, have the code file on-line and use the eXecute command from the Main Command Level to run the utility. When prompted: "Execute what file ?", type:

`ACCESS:TAPEBKUP` [Return] or [Enter]

The following prompt appears on your CRT.

```
Tapebkup: Medium-copy Verify Certify-tape Quit ?
```

Typing the appropriate letter [M], [V], [C], or [Q] selects the corresponding operation.

The Medium-copy Operation

The Medium-copy operation prompts for source and destination media. You specify the source media by entering the volume specification of one of the logical volumes on the media. For instance, `#11:` is often the first logical volume on a multi-volume hard disc. After one of the disc volumes has been specified, you are shown a listing of all the other logical volumes that will be affected. The specification for the tape media is typically `#41:`

Medium-copy confirms that you have not specified the same media twice, and that the two associated drives are on a shared controller. If not, it aborts the operation.

Medium-copy also checks the medium sizes and gives one of two informative messages for the situations where the destination is a tape, and the tape is not large enough to hold the entire source image. If the source is a 7914 disc, in which case the only method of complete backup is with *two long* tapes and appropriate swapping, you are reminded of the fact. If the source is not a 7914 disc, in which case a complete backup *cannot* be performed, you are advised of this situation, one which you should normally avoid!

At this point, the utility will ask:

```
Are you SURE you want to proceed? (Y/N)
```

Confirm your selections, and respond with [Y] or [N].

If the destination is a tape, you are given the option to automatically verify it after the copy completes. As usual, respond with [Y] or [N].

If the destination is the tape and it has never been certified, it will now go through that process. Tapes must be certified in order to support auto-sparing. Note that the "switch" save operation does not automatically certify tapes before writing to them.

The copy now takes place under control of the device itself. It proceeds at a rate of about 35 Kbytes per second, or roughly two Megabytes per minute. At this rate, copies with a 7908 take about eight minutes, a 7911: about 14 minutes, a 7912: a little over 30 minutes, and a 7914: also a little over 30 minutes per tape, or about 65 minutes total. All errors are reported to the CRT. If the destination is a tape and it is not completely filled by the copy, an end-of-file mark is appended to the valid data.

If the destination is a tape and you opted for auto-verification, the verify occurs at this point. Only the data actually written to the tape is verified, so that time will not be consumed verifying the entire tape if data was copied to only a fraction of it.

The utility allows you several options if some error occurs in the above certify/copy/verify segment. This is primarily motivated by the 7914's two tape backup sequence, but it is also a nice feature for the single tape sequences. Specifically, if an error does occur, you may elect to:

- Retry the same segment on the same tape
- Manually change tapes and retry the same segment with a different, supposably better tape
- Ignore the error and proceed, usually to the next segment of a 7914 two-tape sequence
- Abort the entire sequence

Once a segment attempt has been completed, either because there were no errors or because you elected to ignore them, the utility automatically *initiates* the tape unload sequence. If you have a 7914, the utility then prompts you to change tapes, and proceeds with the second tape's certify/copy/verify segment.

Finally, if the destination is a disc, an automatic full-volume verification is performed.

The Verify Operation

The Verify operation prompts for a media specification, which may be either tape or disc. Like the Media-copy operation, you specify the media by giving the volume ID for one of the volumes on the media. The utility then prints out all associated volumes, and asks for confirmation to proceed. Type [Y] or [N].

If the device is a tape, you are also given the option for the utility to automatically initiate the tape unload sequence after the verify. Respond with [Y] or [N].

The verify performed here always covers the entire medium, even if the medium is a tape with file marks embedded in it. In contrast, the optional verify of a destination tape during the Medium-copy operation verifies only the data just copied to the tape.

As the verify proceeds, the addresses of all unreadable blocks are printed to the screen. The verify is considered to have failed if any are encountered.

If you requested the auto-unload option for a tape, and the verify fails, the utility will *not* unload the tape, in anticipation that you will want to take further action with the tape.

The Certify-tape Operation

Providing this operation separately may seem unnecessary since the Medium-copy operation automatically certifies uncertified tapes before it writes to them. However, it has been included in the utility in case you want to certify one or more tapes without having to copy a disc image to each one at this time.

Another use of this operation is to force re-certification of previously-certified tapes. You would want to do this only if you suspect that blocks on the tape had somehow been spared when they were really OK. This might have happened on a tape drive with dirty heads.

The Certify operation prompts for media specification, confirmation of your choice, and the tape auto-unload option, in the same manner as with the Verify operation. In addition it asks:

```
Re-certify if already certified? (Y/N)
```

Normally, you will want to type [N], so that that certification will be done only if the tape has never been certified before. However, if you really want to force a re-certification of the tape, with the resultant loss of any previous sparing information, type [Y].

Quitting the Utility

Simply terminates the utility program.

Using the File System for Direct Tape Access

Pascal 3.0 (and later versions) does provide you with the capability of directly accessing the tape as you would with any other mass storage device. If one cartridge tape drive is present, it will be assigned as a single LIF volume, unit #41; a second drive will be assigned #42:. The intent of this capability is to allow you to initialize the tape using MEDIAINIT, then using the Filer, transfer files or volume images to it, list its directory, change its volume name, etc.

Note

Much of the need for a tape to be directly accessed by the file-system is eliminated by the existence of the BACKUP utility in Pascal 3.2. See the "Using the Backup Utility" section that appears earlier in this chapter.

You can also use the File System to access the *first* volume of a multi-volume disc image that has been backed-up on tape using TAPEBACKUP. You will not be able to access subsequent logical volumes, nor tapes created by the BACKUP utility, nor in general access the second tape of a 7914 backup, without first restoring the image to the disc.

CAUTION

The cartridge tape drives are intended for use as streaming devices. Thus, using these tapes for direct access and selective back-up, although supported, may cause accelerated wear or damage to the tape drive and tape. In other words, use these tapes only for limited back-up and emergency purposes, not for normal file system calls in user programs or as part of a boot sequence.

When you want to use the tape for selective backup/retrieval versus complete backup/retrieval, you have to be careful how you do it, in order to avoid a couple of common pitfalls. These pitfalls are associated with the inherent characteristics of a streaming tape drive, namely its slow seek times and its inability to start and stop rapidly. Note that you should use only LIF for tapes that are to be accessed like a file system. HFS is not supported and would require an even greater number of accesses to the tape than what is listed below for LIF.

For each file written to the tape using LIF, the following sequence occurs:

1. A seek is performed to the very beginning of the tape to scan the directory
2. The entire directory is scanned, one block at a time
3. A seek is performed somewhere "out in the middle" of the tape to write out the file body
4. A seek is performed back to the beginning of the tape to update the directory

With this information at hand we now discuss two general rules.

Avoid Large Directories on the Tape

Considering that streaming tapes like these can't stop and start between blocks, but actually coast to a stop, back up, and take a running start at the next block, you can see that scanning a large directory one block at a time will be a painfully slow process. In addition, it accelerates wear on both the tape and the tape drive.

What constitutes a "large" directory? Ultimately, you will have to decide, but the following data should aid you in making your decision. On a tape with a LIF directory, the first block will contain the LIF volume label and sixteen directory entries. Each block thereafter can contain thirty-two directory entries. Thus, the logical breakpoints in directory sizes are 16, 48, 80, ... 16+32N. MEDIAINIT and the Filer's zero command default to 80 directory entries; it is generally recommended that you not go above this size.

Avoid Transferring Numerous Small Files

Considering that each seek on the tape may take up to tens of seconds, you can see that if you transfer numerous small files, you will probably spend a high percentage of your time seeking back and forth on the tape, and a very small percentage of your time actually transferring data.

Volume Backup

If you are not using the BACKUP utility, another excellent way to efficiently back up numerous small files is to keep all of them on a single logical volume of a LIF disc, and then back up the entire logical volume in one operation. Two previously seldom-used capabilities of the Filer are volume-to-file and file-to-volume transfers; they provide the key mechanism.

A volume-to-file transfer uses an entire logical volume as the source and saves its complete image as a single file on the destination volume. For example, from the Filer you type [F] to specify a file copy, then type:

```
V11:,#41:VOLBACKUP  [Return] or [Enter]
```

to save the entire image of V11: to a file named VOLBACKUP on volume #41, the tape. While a volume image is in a file, the files within the volume are inaccessible, at least to the average user. To make the files within the volume accessible again, you have to transfer the volume image back to a suitable volume, which is usually the one it originally came from, but need not be as long as it has enough room.

Note that because HFS does not support "soft" volumes, this strategy is no better than an image backup using TAPEBKUP, in terms of storage efficiency and speed. Even if you prefix a unit "down" an HFS directory path, then copy that "volume" to a file, the whole contents of the disc gets copied. This technique is also not useful with SRM volumes.

A file-to-volume transfer uses a single file as the source and restores it as a logical volume on the destination volume. For example, from the Filer you type [F] to specify a file copy, then type:

```
#41:VOLBACKUP,#11:  [Return] or [Enter]
```

to restore the file VOLBACKUP, which we assume is a volume image, to its original place. Note that whatever was on #11 is about to be completely overwritten, so the Filer warns you of this, and asks for your confirmation before proceeding.

Advantages to Selective Backup and Retrieval

Even considering the known pitfalls, selective backup/retrieval to the tape with the Filer is a valuable capability. Here are some advantages:

- You can back up only the files/volumes which changed since the last backup, possibly saving time and the amount of media required for backup.
- You can use the CS80 tape to back up files/volumes from all Pascal-supported mass storage devices except SRM and HFS volumes.
- You can interchange data with other HP machines that support LIF on DC600 tapes.
- A single tape can hold many revisions of the same file/volume, for instance during program development. All revisions of the file/volume must be named uniquely, of course.

Note that many of these advantages can be said for using the BACKUP utility. A disadvantage to the volume technique is that it requires a LIF soft volume to exist for the sole purpose of containing the files to be backed up.

HFS Setup and Utilities 21

Setting Up An HFS System

An HFS system can be implemented on almost any local storage media, however it is assumed that, in nearly all cases, implementation will take place on *hard discs* as these are currently the most popular choice of external mass-storage device. HFS may not be a suitable choice for small-capacity storage devices, such as flexible discs or bubbles, since the HFS overhead becomes a larger percentage of the media's capacity.

General Procedure

The basic method of installing HFS requires that the disc be recognized by the system to be of HFS format. To enable the system to access HFS volumes, the module `HFS_DAM` must be added to `INITLIB`. If you have not already done this, use the information supplied in this chapter to perform this task before you go any further. (Additional information regarding HFS can be found in the "File System" chapter and "Special Configurations" chapter of this manual and also in Appendix E of the *Pascal Users' Guide.*)

The following process should be followed:

- Execute `HFS_DAM.` from HFS1: (HFS: on double sided) or copy the module HFS_DAM from file "HFS_DAM" into `INITLIB` and reboot. (See "Adding Modules to INITLIB" in this manual.)
- If the media contains files or valuable data then a backup should be made of the data. See the `BACKUP` utility description in Chapter 19, Non-Disc Mass Storage and Backup Utilities.
- If the media has never been used before, it should be initialized using `MEDIAINIT.CODE`. If the disc previously contained data this step is not necessary.
- Execute the `MKHFS` program as described in the "The MKHFS Utility" section in this chapter.
- Run `TABLE` again. This causes the disc to be recognized as one of HFS format.
- Using the `BACKUP` utility program, transfer the saved data back to the HFS disc.
- If you wish to boot the Pascal Workstation from your new HFS disc, you must create the necessary directories, copy the system files to the disc, and install the boot file on the disc using the OSINSTALL utility, described later in this chapter.

The MKHFS Utility

The make HFS utility (MKHFS) performs a "zeroing" function for HFS discs that is similar to the Filer's Zero command for LIF discs. Before a disc can be used as an HFS disc, it must have an empty hierarchical file structure written on it. Use the MKHFS utility to create the necessary structure on the disc.

Using MKHFS

It is assumed that the data on the disc has been copied onto a different medium using `BACKUP.CODE` and that, if the disc has not been used before, it has been initialized. For the example we will convert unit #11: to HFS. Note that any LIF soft volumes on this disc (e.g., unit #12: etc.) will be destroyed. This is because HFS uses the entire disc and not just a part of the disc.

1. Insert the HFS: or HFS1: disc into a flexible disc drive. (This is not necessary if the System is still resident on another mass storage device).
2. Press [X] and the following prompt will appear:

   ```
   Execute what file?
   ```

3. Execute the `MKHFS.CODE` utility program by typing the volume name followed by the program name. For example:

   ```
   #3:MKHFS [Return]
   ```

   The program will be loaded and will start to execute.
4. The following prompt appears:

   ```
   MKHFS [Rev 3.2 2/15/87] 21-Jan-87 10:15:59

      Copyright 1987 Hewlett-Packard Company.
               All rights reserved.

   Volume ID?
   ```

   Enter the volume number for your disc. For example:

   ```
   #11 [Return]
   ```

   If you wish to print the information produced by this utility, which could be useful if you need to run the file system checking routine HFSCK at a later date, type:

   ```
   #11,#6 [Return]
   ```

   Alternatively, if you wish to send the information to a file, type the name of the file. For example:

   ```
   #11,#4:LOGFILE.ASC
   ```

   The utility will then respond with something similar to:

```
Device: HP7946 fixed disc, 1400,0,0
Logical unit #11 - 'MYSYS:'
WARNING: the initialization will also DESTROY:
 #12 <no dir>
 #13 <no dir>
 #14 <no dir>
  .
  .
  .

Change or examine default parameters? (y/n)
```

5. The default parameters should not normally need to be changed, but if you wish to view or alter them, pressing [y] will cause the screen to clear and the default parameter values to be displayed for editing. If you do not wish to change the default values, type [n] and proceed to step number 6. The display should look similar to this:

```
MKHFS [Rev 3.2 2/15/87]   21-Jan-87 10:15:59

D  Device size in 1Kb blocks:     55000
H  HP-UX swap space, 1Kb blocks:  0
S  Sectors (1Kb) per track:       4
T  Tracks per cylinder:           2
B  Block size in bytes:           8192
F  Fragment size in bytes:        1024
C  Cylinders per group:           default
M  Minimum percent free blocks:   10
R  Rotations per second:          60
I  data bytes per Inode:          2048

Y  Yes, continue
N  No, quit without changing the disc

command?
```

Note

Some SCSI discs do not report enough information about the disc to exactly determine the "sectors per track" and "tracks per cylinder" parameters. In this case, MKHFS will use a heuristic to determine a best guess. It is highly recommended that the "sectors per track" and "tracks per cylinder" parameters **not** be modified.

6. The following prompt will appear:

```
Are you SURE you want to overwrite the disc? (y/n)
```

If you press [y], the disc will be converted to the HFS format and various messages will be printed to the screen to confirm that the program is executing. In particular, you should note the *block numbers* which are given for the superblocks as these numbers could potentially be useful if you should ever need to use the file system checking utility, HFSCK.CODE. It is also possible to list the information on your printer or a file, as

described in step 4. For more information on HFSCK.CODE, see the description later in this chapter.

Finally, the main command menu should appear, signifying that the utility has finished.

A typical output (in this example, a double-sided microfloppy disc initialized with 1K-byte sectors) could be:

```
#3:
770 sectors in 77 cylinders of 2 tracks, 5 sectors
0.8Mb in 5 cyl groups (16 c/g, 0.16Mb/g, 64 i/g)
Alternate superblocks at:
 16, 184, 336, 504, 656,
```

7. It is now necessary to run the **TABLE** program to make the System recognize the HFS disc. Make sure that the TABLE program is on line, press [X] and type:

   `BOOT:TABLE.` and press [Return]

8. If your disc previously contained data, which you saved using **BACKUP**, now is the time to use the same utility to restore these files to your HFS disc. If your disc had not previously been used, you can now create your desired directory structure using the appropriate commands in the Filer.

If you want to be able to boot from this HFS disc you should make the following changes:

- Use the Filer to create the necessary system directories (`/WORKSTATIONS/SYSTEM` – see the section "Re-Naming the BOOT: Files" in Chapter 18, Special Configurations for the possible names of these directories).
- Copy the system code modules to the system directory using the Filer; i.e.

  `INITLIB` (Must be the correct one for your computer and include the HFS_DAM module; see Chapter 18, "Adding Modules to INITLIB")
  `TABLE`
  `STARTUP`

  and

  `ASSEMBLER`
  `COMPILER`
  `EDITOR`
  `FILER`
  `LIBRARIAN`
  `LIBRARY`

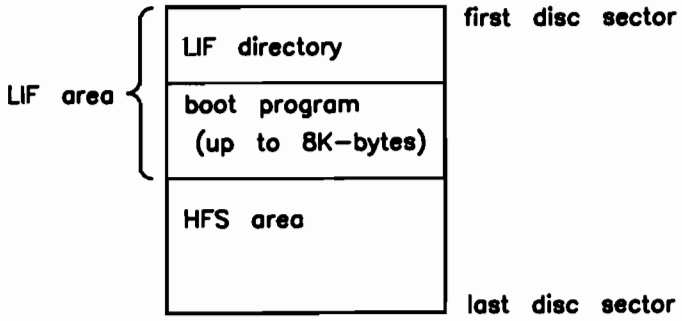
  Plus any other supplied code you wish to access by way of the system volume.
- Use the Filer to copy the **SYSTEM_P** file to the **root** directory (/) of the disc.
- Use `OSINSTALL.CODE` (described in the next section) to install SYSTEM_P in the boot directory.

OSINSTALL Utility

This utility installs boot files on HFS discs. Boot files may be installed, checked and removed using this utility, and a limited *reordering* of the files within the directory is permitted.

HFS Booting Overview

HFS discs are "backward-compatible" to the boot ROMs used in Series 200 and 300 computers. Since the original boot ROMs were created before there were HFS discs, an HFS disc must appear to the boot ROM as something the boot ROM can understand. Thus, each HFS disc has a small LIF-style area at the beginning of the disc.



The boot ROM can only read the the **LIF area**. At boot time, the boot ROM displays all of the names that are in the **LIF directory**. When a name is chosen, for example `SYSTEM_P`, the **boot program** (which does know how to read HFS discs) is loaded, and attempts to find a program of the same name in the root (/) directory of the HFS disc. If the boot program finds the file and the file is of the right format (HP-UX's "`a.out`" format), the file is loaded and booting continues "as usual".

The one exception to this "linkage-by-name" is the case of the file whose name is "`SYSHPUX`". When "`SYSHPUX`" is booted, the boot program looks in the root directory, not for "`SYSHPUX`" but for "`hp-ux`". This is a convention that OSINSTALL also understands.

The boot program cannot load `.SYSTM` files. It requires files, such as `SYSTEM_P`, to be in `a.out` format. OSINSTALL converts "`.SYSTM`" files to "`a.out`" format during installation. The Filer's Extended listing shows "`a.out`" files as type "`Hp-ux`" (t-code -5813).

OSINSTALL Overview

OSINSTALL can reformat a "`.SYSTM`" file (shown by the Filer as type "`System`") located in the root directory of an HFS disc, into the correct format for the boot program, and create an entry for it in the LIF boot directory. The LIF entry will be a "`.SYSTM`" file.

In addition to the "Install" function, OSINSTALL can list the bootable files (the "Check" function), remove a bootable file (the "Remove" function), change which boot file is "first" to be booted automatically (the "Order" function), and can remove all boot files while restoring the boot program (the "Zero" function.)

Using OSINSTALL

The utility is called `OSINSTALL.CODE` and can be found on the HFS: or HFS2: disc. (HFS: for double-sided system discs, HFS2: for single-sided).

To execute the utility from the Main Command Level, make sure the HFS boot disc is on-line, and Prefixed to its root "/" directory. Also make sure the HFS: volume (or HFS2: volume) containing the OSINSTALL utility is on-line. Then press [X] and type:

```
HFS:OSINSTALL [Return]
```
or
```
HFS2:OSINSTALL [Return]
```

The program will be loaded and executed. The following menu will appear:

```
OSINSTALL: Check Install Order Remove Quit Zero ?
```

The system is now waiting for an input. [Stop] or [Q] will return control to the Main Command Level. The options available to you are:

[C] To check the consistency of a boot directory and the boot files on an HFS disc.

[I] To install a ".SYSTM" or "a.out" file as a bootable file on an HFS disc.

[O] To change the order of the files in the LIF boot directory, giving the capability of determining which boot file will be automatically selected if the Boot ROM is not given an input at boot time.

[R] To remove a directory **entry** from the LIF boot directory.

[Q] To quit the OSINSTALL utility.

[Z] To create an empty LIF boot directory, or to recover a corrupted directory such that the disc can be used as a bootable medium again.

The individual options are covered in more detail below.

Check

Depending on the input given, this option will check one of the following:

- A single boot file on a specified volume.
- All boot files on a specified volume.

The necessary input is a volume name, a volume name and file name, or just a file name. Note that wildcards are NOT supported. Also note that for the file "hp-ux" (the boot file supplied with HP-UX systems), you should specify the name "SYSHPUX" not "hp-ux" should you ever need to reinstall the HP-UX boot file.

For each file checked, there are four possible results which can be output by the utility:

`bootable` — The file is in the correct state, i.e. in the correct format and bootable.

`not bootable` — The system file in the root directory is not in **a.out** format.

`not a LIF boot file` — The LIF directory entry is not of bootable type (not `.SYSTEM`).

`not found in hfs`*n*`:/` — There is no corresponding file in the root directory (in "/" for the given volume.)

Invoking the Check Option

To invoke Check you should press [C] when in the OSINSTALL menu. The following prompt will then appear on the screen:

```
Volume:file to check (in boot area) ?
```

The following example responses should clarify the way in which this option works:

`#43:SYSTEM_P`	This will check the file `SYSTEM_P` on volume `#43`.
`#11`	This will check all files on volume `#11`.
`SYSTEM_P`	This will check the file `SYSTEM_P` on the default volume.

Note that directory paths are not supported. The volumes ":" and "*" are understood. The output provided by the utility will be similar to that shown below. How many files are checked naturally depends on the input given. An example output might show:

```
SYSTEM_P    bootable
SYSTEM_H    bootable

SYSTEM_L    not found in hfs11:/
SYSTEM_Z    not a LIF bootfile (not .SYSTM)
```

Problem files can be eliminated by using the Remove command in this utility to get rid of the unwanted boot files.

CAUTION

HFS flexible discs present something of an anomaly since both the LIF boot area and the "real" HFS disc are normally accessible via the file system. For example, the LIF boot area on an HFS flexible disc appears at unit #3, while the "real" HFS disc appears at unit #43:.

It is important to never access the LIF boot area except by using the OSINSTALL utility. Damage to the HFS structure may occur if the LIF boot area is modified.

Install

Selecting this option gives you the facility to install a system file and make it bootable.

The file which you wish to install must be in the root directory of the volume where you want the bootable file to reside, and the access rights for the file and directory should be sufficient for the utility to perform the necessary tasks; see Chapter 5 for HFS access right specification. The HFS unit should be prefixed to its root (/).

Invoking the Install Option

To invoke this option press I when the OSINSTALL menu is displayed. This will cause the following to appear on the screen:

```
Volume:file to install (from root directory on HFS) ?
```

The appropriate response should now be given. The volume name should be prepended to the file name if the file is not in the default volume. Pathnames are not supported. The utility will now attempt to perform the installation, displaying the following message on successful completion:

```
hfs11:SYSTEM_D installed
```

The OSINSTALL menu will then be displayed.

The maximum number of entries permissible in the LIF boot directory is normally eight, however, the OSINSTALL utility will convert the directory to hold sixteen entries if an option other than Check or Quit is chosen. Should you attempt to install a seventeenth system, the following error message will be given:

```
Error: no room in directory
```

Order

If there is no intervention from the user at boot time, Series 200/300 computers will boot the first boot file found on the lowest addressed mass storage unit. This option in the OSINSTALL menu gives you the facility of moving a selected system file (e.g. SYSTEM_D) to the *beginning* of the LIF boot area directory so that this file will be the one automatically selected at boot time. (Assuming the volume is the lowest addressed mass storage unit.)

Invoking the Order Option

From the OSINSTALL menu, press [O]. The following display will appear:

```
Volume:file to move to first position (in boot area) ?
```

Enter, for example: `SYSTEM_D`

You should now type in the filename you wish to move, with its corresponding volume identifier. For the above input example, when the option is complete, it will display the following message and control will be returned to the OSINSTALL menu where you may choose another option or quit.

```
SYSTEM_D now in first directory position
```

Remove

This option removes system file entries from the LIF directory of an HFS disc. It does **not** remove the actual system file from the root directory. (That can be done with the Filer, if you have the correct file permissions.) No wildcards can be used.

Invoking the Remove Option

To invoke this option, press the `R` key. This will cause the following to be displayed:

```
Volume:file to remove (from boot area) ?
```

Assuming a legal input is given the option will remove the file entry and display:

```
SYSTEM_D removed from LIF boot directory.
```

Should this be the last remaining system file in the directory, the following warning will be given **before** the file is removed.

```
This will remove the last bootable file.
Are you SURE you want to proceed? (Y/N)
```

Zero

This option is similar to the Zero command in the Filer subsystem. Its operation is limited to the LIF boot directory of an HFS disc. It is expected that this option will only be used when the LIF directory of an HFS disc has in some way become corrupted, or there is a need to "start from scratch" with the boot area.

Invoking the Zero Option

This option is invoked by pressing `Z` when in the OSINSTALL menu. The following display will appear:

```
Volume (boot area) to zero ?
```

After the volume number has been entered the option will perform the task of zeroing the disc and will confirm that the task is complete by displaying, for example:

```
Volume hfs3: LIF boot directory zeroed.
```

The HFSCK Utility

The Hierarchical File System Check (HFSCK) utility checks and repairs broken HFS format file systems. It is recommended that this utility be used on a regular basis if you are using HFS mass storage devices regularly. It is worth considering doing a file system check each time you boot the system by putting the necessary invocation commands in the AUTOSTART file, including a file listing of the directory `lost+found` after the checking is complete, to see if there are any troublesome files. This check does take a considerable amount of time.

Why Should I Need To Run This Utility?

If an HFS disc is switched off during an I/O operation, the system is powered down without being at the Main Command Level, or a program is written which inadvertently corrupts the operating system and thereby the control of the file system, the file system could reach a state which is not irretrievable but is nevertheless in need of some repair. Normally, HFS is expected to be nearly as reliable as LIF. This utility is designed to be used to perform the repair work.

A directory with the name `lost+found` must exist in the root directory of the file system to be examined **before** HFSCK is executed. When an HFS disc is created with MKHFS, the `lost+found` directory is automatically created and 8K-bytes of space allocated to it. Do not remove or reduce the size of the `lost+found` directory because HFSCK needs it for proper operation. HFSCK puts any problem files it finds in the `lost+found` directory. After running the utility you should examine any files placed in the directory and move them to where they belong or remove them. The contents of `lost+found` should be cleared before running the utility again.

As the size of your file system increases, so does the amount of RAM required to run the utility. There is no fixed rule, but you should use the utility before P-loading files, creating memory volumes, and performing other memory-consuming operations.

Each phase of the utility invokes an different examination of the file system. Successive phases check: blocks and sizes, pathnames, connectivity of files to inodes, reference counts, and the free-block map.

When an inconsistency is detected, the error condition is reported by displaying pertinent information, or a question, on the screen. In the following section, each error message and the possible responses are presented.

Note that some items are mentioned in the following discussions that are foreign to the Pascal Workstation (e.g. pipes, fifos, etc.). These can only be created by HP-UX (or appear to be on the system due to some accidental damage to the file system integrity). Since HFSCK must work on discs shared by the Pascal Workstation and HP-UX, the utility "understands" these special items. If your disc is being used by both the Pascal Workstation and HP-UX, it may be best to have the HP-UX "superuser" check the disc with *fsck*, which is roughly equivalent to HFSCK.

CAUTION

If your Pascal Workstation shares a disc with a 6.5 or later version of HP-UX and access control lists (ACLs) are in use, the use of HFSCK will cause the ACL information to be lost. If you fall in this category and suspect that your disc may be corrupt, boot HP-UX and let the HP-UX `fsck` program check the disc.

Invoking the HFSCK Utility

This utility program is supplied on the HFS: or HFS2: disc as `HFSCK.CODE` and is invoked from the Main Command Level by pressing [X] and entering:

`HFS:HFSCK` [Return]

or

`HFS2:HFSCK` [Return]

The utility will be loaded into memory and executed, the following display appearing on the screen:

```
HFSCK [Rev 3.2]  1-Jan-87 17:26:40

   Copyright 1987 Hewlett-Packard Company.
           All rights reserved.

Volume ID?
```

The volume number, or name, for the device which contains the HFS to be checked should now be entered. Providing the input is valid the utility will continue by asking:

```
Report only, ask always, or normal confirmation (r/a/n)?
```

- Report only – reports disc problems but does not attempt to repair them.
- Ask always – attempts to repair the disc and asks for confirmation of each change.
- Normal confirmation – attempts to repair the disc by automatically fixing the *harmless* problems and asking for confirmation with the others.

Finally, the third question allows you to specify an alternate copy of the superblock in case the standard one has become corrupted. The input required here is a block number which corresponds to a superblock on the disc. These numbers were displayed when the MKHFS utility was executed; i.e. when the disc was initially formatted for HFS. A null input, generated by simply pressing [Return], will cause the standard superblock to be used. The utility will prompt:

```
Alternate superblock block number (RETURN for default)?
```

If the file system is in good order, you can expect to see a display similar to this:

```
** #43
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
7 files, 14 used, 177 free (17 frags, 20 blocks)
```

The file system may require some "tidying and repairing" by the utility. Depending on the mode chosen, you could be asked to answer some *confirmation requests.* These requests are discussed below.

HFSCK Confirmation Requests

This section covers the possible confirmation requests that can arise during each phase of the checking process and the corresponding valid responses with their consequences. In *normal confirmation mode*, some of the following questions will be answered automatically with a "yes". In this case, HFSCK will automatically fix problems that do not usually require any user interaction. This is typically the *safest* way and is the mode which is recommended. If you choose a mode other than normal confirmation and you give an answer other than "yes" to a confirmation request, the utility will not necessarily be able to repair the file system. Unless you have a clear understanding of the way in which the file system operates, it is recommended that you do not follow this route as you could potentially do more damage than repair.

In the following section, `[Y]` means that the question is automatically answered "yes" in normal confirmation mode.

Phase 1 - Check Blocks and Sizes

This phase checks the block numbers and file size in each inode. The block numbers must be sensible and agree with the size. No block may be in more than one file.

`HOLD BAD BLOCK?`

An inode has an illegal type field of the sort created by the UNIX[1] (but not HP-UX) badblk utility.

Y : Convert this to a regular file containing the one bad block.

N : Clear the inode.

```
[Y] NON-ZERO READER/WRITER COUNTS ON PIPE I=xxx
CORRECT?
```

A "first-in-first-out" (fifo) has non-zero reader or writer counts.

Y : Reinitialize fifo.

N : Leave inode as it is.

```
[Y] BAD DIRECT ADDRESS, SHOULD BE ZERO: inode.di_db[xxx] = yyy
CORRECT?
```

An inode has a direct address beyond the limit imposed by the file size.

Y : Zero the address.

N : Leave it as it is.

```
BAD INDIRECT ADDRESS: IND BLOCK xxx[yyy] = zzz
CORRECT?
```

An inode has an indirect address beyond the limit imposed by the file size.

Y: Zero the address.

N: Leave it as it is.

```
COULD NOT CHECK INDIRECT BLOCKS
CLEAR?
```

HFSCK could not check the inode's indirect blocks.

Y: Clear the inode.

N: Leave it as is.

```
LINK TABLE OVERFLOW.
CONTINUE?
```

There are more than 500 files with negative or zero link count.

Y: Keep going. HFSCK may be unable to produce a clean file system, so you should run it again.

N: Exit HFSCK.

```
[Y] INCORRECT BLOCK COUNT I=xxx (yyy should be zzz)
CORRECT?
```

The block count in inode xxx is yyy instead of the zzz required by the inode size.

Y: Change the block count to agree with the size.

N: Leave it alone.

```
UNKNOWN FILE TYPE I=xxx ...
CLEAR?
```

HFSCK does not understand the inode contents. This arises in several situations:

The inode type is not recognizable.

The size is negative.

There are indirect disk addresses beyond the limit implied by the size.

Y: Clear the inode.

N: Leave it alone.

```
UNREF PIPE I=xxx
CLEAR?
```

An inode marked unallocated has some numbers in places where a pipe might have some.

Y: Clear the inode.

N: Leave it alone.

```
PARTIALLY ALLOCATED INODE I=xxx
CLEAR?
```

An inode marked unallocated has some disk addresses.

Y: Clear the inode.

N: Leave it alone.

```
xxx BAD I=yyy
```

Inode yyy contains an unreasonable disk address xxx. The address is too large for the file system, or indicates a block in an area where data blocks shouldn't be. There is no confirmation; this is just an error report.

```
EXCESSIVE BAD BLKS I=xxx
CONTINUE?
```

There are more than 10 bad block numbers in the file.

Y: Skip this pass. HFSCK may be unable to produce a clean file system, so run it again.

N: Exit HFSCK.

```
xxx DUP I=yyy
```

Inode yyy contains a disk address xxx found in another inode. There is no confirmation; this is just an error report.

```
EXCESSIVE DUP BLKS I=xxx
CONTINUE?
```

There are more than 10 duplicate block numbers in the file.

Y: Skip this pass. HFSCK may be unable to produce a clean file system, so run it again.

N: Exit HFSCK.

```
DUP TABLE OVERFLOW.
CONTINUE?
```

There are more than 100 duplicated blocks.

Y: Keep going. HFSCK may be unable to produce a clean file system, so run it again.

N: Exit HFSCK.

Phase 1a - Rescan for more DUPS

This phase only occurs if phase 1 found some duplicate blocks. This will result in some more "xxx DUP I=yyy" messages being output. HFSCK must rescan because when it discovers that a block is duplicated, it has forgotten which inode contained the first reference to the block.

Phase 2 - Check Pathnames

HFSCK traverses the directory tree, checking the directories for consistent entries, presence of "." and "..", and possibly bad numbers.

```
ROOT INODE NOT DIRECTORY
FIX?
```

The inode for "/" is not a directory.

Y: Change its type and try to interpret the contents as a directory.

N: Exit HFSCK.

```
DUPS/BAD IN ROOT INODE
CONTINUE?
```

The inode for "/" contains bad or duplicated blocks.

Y: Try to keep going.

N: Exit HFSCK.

```
ZERO LENGTH DIRECTORY I=xxx OWNER=yyy ...
REMOVE?
```

The directory whose inode is xxx has size 0.

Y: Clear the inode.

N: Leave it alone.

```
DIRECTORY TOO SHORT I=xxx OWNER=yyy ...
FIX?
```

The directory size is less than the minimum (big enough to hold '.' and '..').

Y: Change the size to the minimum.

N: Leave it alone.

```
DIRECTORY CORRUPTED I=xxx OWNER=yyy ...
FIX?
```

An entry for the directory whose inode is xxx is not consistent, or the directory size is not a multiple of the size of a directory entry.

Y: Correct the entry.

N: Leave it alone.

```
BAD INODE NUMBER FOR '.' I=xxx OWNER=yyy ...
FIX?
```

The entry for '.' does not contain the inumber of the directory.

Y: Correct the entry.

N: Leave it alone.

```
MISSING '.' I=xxx OWNER=yyy ...
FIX?
```

There is no entry for '.' in the directory.

Y: Add entry for '.'.

N: Leave it alone.

```
BAD INODE NUMBER FOR '..' I=xxx OWNER=yyy ...
FIX?
```

The inumber for the '..' entry is not the same as the inumber of the parent of xxx.

Y: Correct the entry.

N: Leave it alone.

```
MISSING '..' I=xxx OWNER=yyy ...
FIX?
```

There is no entry for '..'.

Y: Add entry for '..'.

N: Leave it alone.

```
EXTRA '.' ENTRY I=xxx OWNER=yyy ...
FIX?
```

The directory has more than one entry for '.'.

Y: Remove all but the first entry for '.'.

N: Leave it alone.

```
EXTRA '..' ENTRY I=xxx OWNER=yyy ...
FIX?
```

The directory has more than one entry for '..'.

Y: Remove all but the first entry for '..'.

N: Leave it alone.

```
I OUT OF RANGE I=xxx OWNER=yyy ...
REMOVE?
```

An entry in the directory has an impossible inumber.

Y: Remove the entry.

N: Leave it alone.

```
UNALLOCATED I=xxx OWNER=yyy ...
REMOVE?
```

An entry in the directory has an inumber for an inode that is unallocated.

Y: Remove the entry.

N: Leave it alone.

```
DUP/BAD I=xxx OWNER=yyy ...
REMOVE?
```

An entry in the directory has an inumber for an inode with duplicate or bad blocks.

Y: Remove the entry. This can be very dangerous! The inode may have duplicate blocks (same block as in another file), but it may be the OTHER inode that is bad, not this one.

N: Leave it alone.

Phase 3 - Check Connectivity

This phase finds directory trees not connected to "/", and reconnects them to "lost+found".

```
[Y] UNREF DIR I=xxx OWNER=yyy ...
RECONNECT?
```

The directory is not connected to "/".

Y: Reconnect it to "lost+found" if possible. Note that HFSCK will not automatically increase the size of the "lost+found" directory. It will only insert the orphan directory if there is an empty slot.

N: Leave it alone.

Phase 4 - Check Reference Counts

This phase checks the link counts on all inodes and adjusts them if necessary. If an inode is unreferenced, it clears it or links it into lost+found.

```
[Y] UNREF FILE I=xxx OWNER=xxx ...
[Y] UNREF PIPE I=xxx OWNER=xxx ...
RECONNECT?
```

The file has no references in any directories.

Y: Reconnect it to "lost+found" if possible. Note that HFSCK will not automatically increase the size of the "lost+found" directory. It will only insert an orphan file if there is an empty slot.

N: Leave it alone.

```
[Y] UNREF DIR I=xxx OWNER=yyy ...
[Y] UNREF FILE I=xxx OWNER=yyy ...
[Y] UNREF PIPE I=xxx OWNER=yyy ...
[Y] BAD/DUP DIR I=xxx OWNER=yyy ...
[Y] BAD/DUP FILE I=xxx OWNER=yyy ...
[Y] BAD/DUP PIPE I=xxx OWNER=yyy ...
CLEAR?
```

The inode is not referenced, and it is not a regular file, otherwise it has size 0 or you did not want to reconnect it into lost+found.

Y: Clear the inode.

N: Leave it alone.

```
[Y] LINK COUNT FILE I=xxx OWNER=yyy ...
[Y] LINK COUNT DIR I=xxx OWNER=yyy ...
[Y] LINK COUNT filename I=xxx OWNER=yyy ...
COUNT a SHOULD BE b
ADJUST?
```

The link count in the inode is incorrect.

Y: Correct it.

N: Leave it alone.

```
[Y] FREE INODE COUNT WRONG IN SUPERBLOCK
FIX?
```

The free inode count is wrong.

Y: Fix it.

N: Leave it alone.

Phase 5 - Check Cyl groups

This phase examines the cylinder groups and corrects bad information, if found.

```
[Y] FREE BLK COUNT(S) WRONG IN SUPERBLOCK
FIX?
```

The free block totals in the superblock are wrong.

Y: Fix it.

N: Leave it alone.

```
[Y] BAD CYLINDER GROUPS
FIX?
```

The cylinder group information does not match reality.

Y: Enter Phase 6.

N: Leave them alone.

```
EXCESSIVE BAD BLKS IN BIT MAPS
CONTINUE?
```

There are too many bad blocks in the cylinder group bit maps.

Y: Enter Phase 6.

N: Exit HFSCK.

Phase 6 - Salvage Cylinder Groups

This phase reconstructs the cylinder group information.

Porting to Series 300 22

Introduction

This chapter focuses on one objective: making Pascal programs written for Series 200 computers run on Series 300 computers. This process is known as "porting" programs.

Who Needs this Information?

This chapter is directed toward you if you have existing software for Series 200 machines — programs developed by either someone else or yourself. Therefore, it will be of little or no use to you if you are just beginning to develop software for a Series 300 computer.

Methods of Porting

Here are several methods of porting Series 200 software to Series 300 machines:

- Just load it into a Series 300 computer — with no modifications — and run it.
- Write and run a program that properly configures the Series 300 computer for the program.
- Make your Series 300 computer emulate a Series 200 Model 217 computer (by installing a HP 98546A Compatibility Video Card Set), and then run your *unmodified* Series 200 object code on it.
- Modify your Series 200 Pascal source code, re-compile it on the Pascal 3.2 system, and then run it on a Series 300 computer.

Each method has a slightly different set of requirements for its use, as described subsequently.

Chapter Organization

This chapter is organized according to the above strategies. It consists of the following sections:

- Description of enhancements provided by Series 300 computer hardware
- When and how to just load and run the program
- When and how to use a configuration program
- When and how to use the compatibility card set
- When and how to modify the program's source code

Description of Series 300 Enhancements

Acquiring a general understanding of the enhancements to Series 200 computers provided by Series 300 computers will help you to choose a porting method.

Areas of Change

Series 300 computers have enhancements in the following areas:

- Many choices of processor, display, and human interface boards:
  - Seven new displays (including a separate, high-speed display controller)
  - Three new processors: MC68010, MC68020 (with MC68881 math co-processor), and MC68030 (with MC68882 math co-processor)
  - A 32-bit address bus and 32-bit data bus in the Model 330 and 350
  - Battery-backed, real-time clock
  - RS-232C serial interface (similar to the 98644 serial interface)
  - 46020 or 46021 HP-HIL keyboard (similar to keyboards used with Models 217 and 237, but different from other Series 200 models)
  - Parallel interface on Models 345, 375 and later computers
  - SCSI interface on Models 345 and 375 computers, and built-in SCSI discs on Models 340 and 345
- No ID PROM (not all Series 200 Models had this feature)

Areas that Did Not Change

It will probably be comforting to know that if a feature is not listed above (and discussed in this chapter), then it is the same for Series 300 computers as for Series 200 computers.

It may also be comforting to note that Series 300 computers can use most of the Series 200 accessories and peripheral devices. See the *HP 9000 Series 300 Configuration Reference Manual* for a complete list.

Displays

Series 300 display technology is the most visible area of change from Series 200 computers.

All Series 300 computers utilize bit-mapped alpha display technology, which *combines* alpha and graphics. (Only the Series 200 Model 237 has a bit-mapped alpha display; all other Series 200 models have *separate* alpha and graphics planes.)

The main difference between "non-bit-mapped" and "bit-mapped" alpha displays is most easily described in terms of whether the alpha and graphics planes are on independent planes or are on the same plane.

- With "non-bit-mapped" alpha displays, the alpha plane is *separate* from the graphics plane. You can use the [ALPHA] and [GRAPHICS] keys to turn each plane on. When the alpha display is already on, pressing the [ALPHA] key turns off the graphics display. Similarly, pressing the [GRAPHICS] key while the graphics display is on turns off the alpha plane.
- With "bit-mapped" alpha displays, alpha and graphics are displayed on the same plane; there are no separate alpha and graphics planes.

An effect of bit-mapped alpha is that both alpha and graphics are dominant. In other words, displaying a character on the screen overwrites all pixels within the character cell; the previous contents of those pixels are lost. Also, any scrolling/clearing of the alpha screen will scroll/clear the graphics information on the screen, since they share the same display plane.

With Series 300 computers, you may choose from one of seven displays: both monochrome and color, each available in both medium- and high-resolution versions. Each of these displays requires a different monitor. (Series 200 computers have only one display available for each model.)

- Medium-resolution graphics displays have a default resolution of 512[1] horizontal by 385[2] vertical pixels with DGL (many of the Series 200 graphics displays had 512×390-pixel graphics displays).

  Alpha capabilities of these medium-resolution displays are 26 lines by 80 characters (as opposed to the 25×80-character alpha displays of many Series 200 computers). The character font for medium-resolution Series 300 displays is a 10×10-pixel character in a 12×15-pixel cell. These displays have no blinking mode (except for the alpha cursor), and no half-bright mode.

- High-resolution displays have a default resolution of either 1 024 horizontal by 752[3] vertical pixels, or 1 280 horizontal by 1000 vertical pixels, with DGL.

  Alpha capabilities of high-resolution displays are 48 lines of 128 characters, just the same as on the Model 237. The characters are 6×10-pixel characters in an 8×16-pixel cell, or 8×13-pixel characters in a 10×20-pixel cell. These displays also have no blinking mode and no half-bright mode.

---

[1] All 98542, 98543, 98544 and 98545 displays actually have 1 024 horizontal pixels. However, on medium-resolution displays, pairs of contiguous, non-square pixels are treated by the graphics library (DGL) as one unit in order to make square dots on the screen.

[2] Medium-resolution Series 300 displays have 400 vertical pixels displayed, of which only 385 are used as a default by DGL. You can also have up to 400 by disabling the on-screen echo of the type-ahead buffer (set bit 8 of the DISPLAY_INIT procedure's "control" parameter).

[3] High-resolution Series 300 displays have 768 or 1024 vertical pixels displayed, of which only 752 or 1000 are used as a default by DGL. You can also have up to 768 or 1024 by disabling the on-screen echo of the type-ahead buffer (set bit 8 of the DISPLAY_INIT procedure's "control" parameter).

Processor Boards

Four processor boards are available with Series 300 computers:

- The medium-performance board, features an MC68010 processor (10 MHz clock rate).
- The three higher-performance boards feature an MC68020 processor (16 or 25 MHz clock rate) and an MC68881 floating-point math co-processor.

The 332, 340, 360, and 370 SPUs feature MC68030 processors and MC68882 floating-point co-processors. (The MC68882 is optional on the 332.) Clock speeds range from 16 MHz to 33 MHz from these SPUs.

Series 200 computers have an MC68000 processor with an 8 MHz clock, or a 12.5 MHz clock and "HP-UX memory-management hardware" in products with a "U" suffix, such as a "Model 236U."

The 68010 is a 16-bit virtual memory microprocessor with a 32-bit internal architecture, and a 16 Mbyte (24-bit) address space. The treatment of virtual memory and the virtual machine of the MC68010 is extended in the MC68020, a 32-bit microprocessor with cache, 32-bit data and address buses, 32-bit data paths, and a four Gbyte (32-bit) linear address space. (Note that only 16 Mbytes of address space are available with the Series 200 and Model 310/320 systems, because only 24 of the address bits are implemented in the computer's backplane.) On Models 330, 332, 340, 350, 360, and 370 four Gbytes of address space are available due to the full 32-bit addressing in the backplane.

Both the MC68020 and MC68030 contain an internal 256-byte instruction cache. The MC68030 processor also contains a 256-byte data cache managed similarly to the instruction cache. Each time the microprocessor goes off-chip to fetch opcodes and data, the cache retains the information. Should the need arise to re-execute a recent instruction sequence, the sequence within the cache may still be valid. In this case, the processor reads the instruction information out of the cache without accessing off-chip resources, thus speeding execution. While the MC68020 or MC68030 is executing from the cache, any other bus masters, such as DMA controllers, are free to use the external buses without halting the processor. Pascal does not use virtual memory but does enable the cache if the hardware exists (as on Models 320, 330, 332, 340, 350, 360 and 370).

The MC68020 and MC68030 also have a flexible co-processor interface that allows close coupling between the main processor and co-processors such as the MC68881 and MC68882 respectively. The co-processor, which provides full IEEE floating-point math support, can execute concurrently with the main processor and usually overlaps its processing with the 68020/30's processing to achieve higher performance. The co-processor provides increased performance for floating-point operations, in both speed and accuracy, particularly for the evaluation of transcendental functions.

Battery-Backed Real-Time Clock

The Model 310's processor board and the Model 320/330/350's Human Interface boards have a built-in, battery-backed, real-time clock. However, this clock has a limited range compared to the Series 200 real-time clock; its range is January 1, 1900 through December 31, 1999. (Only Series 200 Models 226 and 236 could have optionally installed battery-backed, real-time clocks. This hardware was included with the HP 98270 Powerfail Option, whose main purpose was to provide power during brown-out or black-out situations.) For Revision 3.22 and later dates stored as 1JAN00 through 31DEC27 are interpreted as January 1, 2000 through December 31, 2027. 1JAN70 through 31DEC99 are interpreted as January 1, 1970 through December 31, 1999. 1JAN28 through 31DEC69 are invalid.

If your program uses the Series 200 battery-backed real-time clock, you may need to modify and re-compile the program's source as described in the subsequent "Modifying a Program's Source Code" section.

Built-In Interfaces

All Series 300 computers have a built-in HP-IB interface, which is the same as the built-in HP-IB interface of all Series 200 computers.

Series 300 computers also feature the following built-in interfaces, which differ slightly from some of their Series 200 counterparts:

- RS-232C serial interface (like the HP 98644 low-cost serial interface).
- HP-HIL keyboard interface (like the one in Models 217 and 237)
- Some Series 300 computers include a LAN, DMA, HP 98625 HP-IB, HP 98265 SCSI, built-in SCSI disc, and parallel interface.

Note that LAN drivers are supplied with the Pascal 3.22 Workstation, but their default select code is the same as the default for the SRM interface. You must change one or the other if you use a Model 330, 340, 350, 360 or 370 with an SRM system.

SCSI and PARALLEL drivers are supplied with Pascal 3.23 and later workstations.

Serial Interface

All Series 300 computers have a built-in serial interface. As with Series 200 Models 216 and 217 built-in serial interfaces, this interface is permanently set to select code 9. However, this interface differs slightly from versions of the Series 200 built-in serial interface (which are like the optional HP 98626 serial interface).

Since the goal of the built-in 98644 is to provide a low-cost serial interface, there are no hardware switches that allow you to specify default values for the following parameters:

- Select code (hard-wired to 9)
- Interrupt level (hard-wired to 5)
- Default baud rate (Pascal system sets default to 2400 baud)

- Default line control parameters (Pascal system sets defaults to 8 bits/character, 1 stop bit, parity disabled).

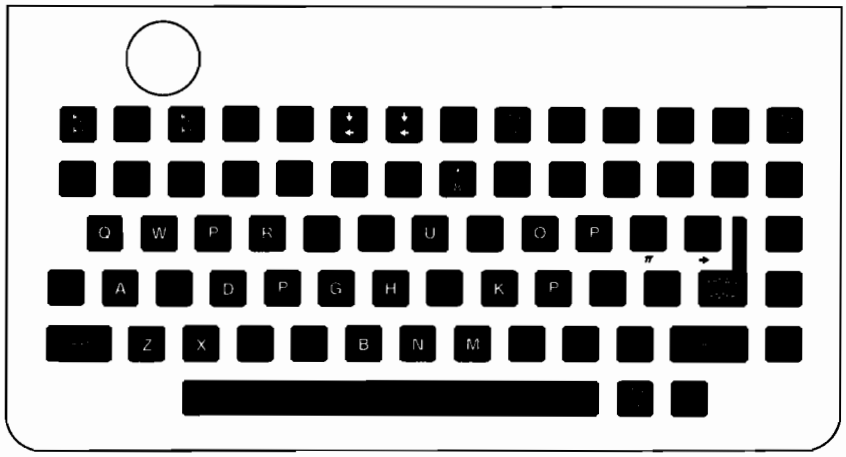
If your program expects any other values for the baud rate and line control parameters, you will have to change them programmatically (select code and interrupt level cannot be modified programmatically). See the subsequent "Using a Configuration Program" section of this chapter for further information.

HP-HIL Keyboard Interface

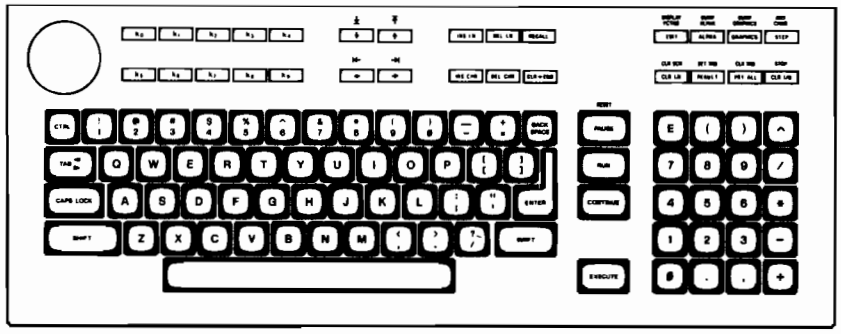
Like the Series 200 Models 217 and 237 computers, Series 300 computers use the HP 46020A or HP 46021A HP-HIL (Hewlett-Packard Human Interface Link) keyboard.

If you are porting existing Series 200 software to Series 300 and have already modified it to run on a Model 217 or 237 computer, then you have already made the adjustments necessary for this keyboard. If not, then continue reading this section.

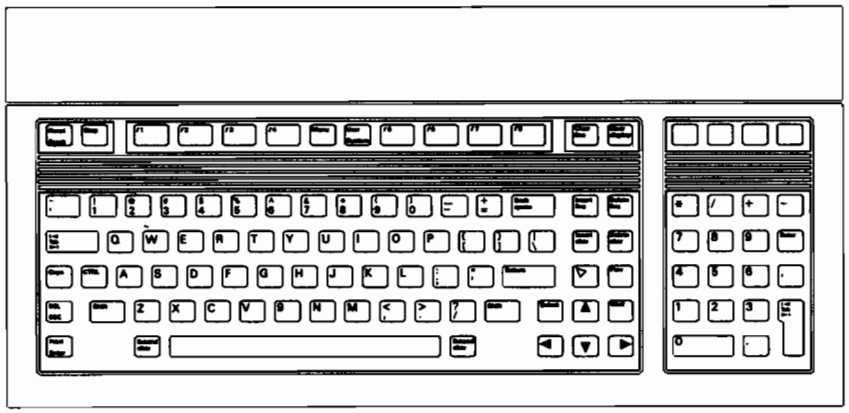
The major human-interface differences between the 98203B keyboard (Models 216, 220, 226, and 236) and the HP-HIL keyboard are in the number and layout of function and system keys.



HP 98203A Keyboard



HP 98203B/C Keyboard



HP 46020A/21A Keyboard

Note that the Series 300 (46020/21) keyboard has only eight "function keys," and lacks some of the system keys on the 98203 keyboard. However, the 46020/21 has *all* of the functionality of the 98203 function and system keys by providing [System] definitions for keys [f1] through [f8]. (Press [System] and then [Menu] on the 46020/21 keyboard to display the system-key labels on the bottom of the monitor screen; default user-key labels are not provided.)

The key mapping is as follows:

- 98203 function keys [k1] through [k8] map into 46020/21 [User] function keys [f1] through [f8]. (The shifted function keys map similarly.)
- 98203 function keys [k0] and [k9] map into 46020/21 [System] function keys [f1] and [f8].
- Other 98203 system keys map into 46020/21 [System] keys (see the key labels by pressing [System] and then [Menu] or [Shift] [Menu]).

Also note that the 98203 keyboards can produce some keycodes that cannot be produced with the 46020/21 keyboard. These key codes are produced by pressing the [EDIT] and [RUN] keys. Thus if the Series 200 program depends upon these keys, the source code must be modified and re-compiled. The topic of trapping key codes with a program is described in the "System Devices" chapter of the *Pascal Procedure Library* manual.

ID PROM

Note that there is no ID PROM available with Series 300 computers, as was the case with many models of Series 200 computers. The HP-HIL ID Module (HP 46084A) is unsupported by Pascal on the Series 300 computers.

Just Loading and Running Programs

This is the most desirable method, since it requires the least amount of work — just load the program into the Series 300 computer, and run it. This section describes when and how to use this method or porting programs.

You can probably port *most* of your programs this way, if they have been written under Pascal 3.0 or later.

There are three different actions you can take, depending on who developed your program:

- If HP developed the program, look in the "Operating Systems and Applications" section of the *HP 9000 Series 300 Configuration Reference Manual.* The manual shows which 3.0 (or later) applications will run on a Series 300 computer using the 3.2 system.
- If another software vendor developed the program, then that vendor should be able to tell you whether or not it will run on a Series 300 computer. (You can also take one of the two actions listed below.)
- If you developed the program, you can do one of two things:
  - Read through the following sections to see whether it requires another porting method.
  - Try running it.

Should Problems Arise

If your program will not run on your Series 300 system, then you may want to consider the following:

- Does it meet *all* of the criteria listed in the subsequent sections?
- Is there sufficient memory in the computer?
- Are all the necessary devices and corresponding device drivers installed?
- Have you fulfilled *all* other requirements listed by the software developer?

If the program still doesn't run, then you may want to call the organization responsible for supporting the program (the programmer, the software vendor, or HP).

Using a Configuration Program

This method involves writing a program that configures the system for your program. This section describes when and how to use this method of porting.

Example of Serial Interface Configuration

Here is an example situation for which you could use this method. Suppose your program depends on reading the following parameters from the configuration switches on the 98626-like, built-in serial interface in a Model 217:

- 4800 baud
- 7 bits per character (with 1 stop bit) and odd parity.

However, there are no such switches on the built-in 98644-like interface in Series 300 computers; instead, the Pascal System gives them the following default values:

- 2400 baud
- 8 bits/character (with 1 stop bit), and parity disabled

One solution is to write and run a short program that selects the desired "non-default" baud rate (4800) and line-control parameters (7 bits, odd parity), and then run the program before running the your application program.

This example program changes the "default" parameters by writing to IOCONTROL registers 21 (baud rate) and 22 (line control).

```
program Serial(input,output);
import general_0,general_1;
begin
  ioinitialize;
    iocontrol(9,21,4800);                 { Baud rate. }
    iocontrol(9,22,binary('11001010'));   { No handshake (bits 7,6)
                                            Odd parity   (bits 5-3)
                                            1 stop bit   (bit  2)
                                            7 bits/char  (bits 1,0)
                                          }
  iouninitialize;
end.
```

You could compile and run this program on the 3.2 system (making sure that the 3.2 IO library is accessible during both compilation and loading[1]). If the RS232 module is not installed, you should install it (with the eXecute command). Then Run the application program, and the serial card will be properly configured.

[1] The easiest way to ensure this accessibility is to put the LIBRARY: disc on-line and then use the Main Level "What" command to specify the **LIBRARY:IO** file as the System Library (with double-sided media, this is the SYSVOL:IO file).

Another solution is to modify the source program to select these parameters. In such case, you could change the "current" parameters by writing to IOCONTROL registers 3 (baud rate) and 4 (line control). However, if the program later resets the interface with any of the following operations:

- IOINITIALIZE
- IOUNINITIALIZE
- IORESET
- IOCONTROL of registers 1 or 14

or if you press the [Stop] key (or [CLR I/O] on the 98203 keyboards), then the values in these registers will be restored to the "default" values currently in registers 21 and 22. See the *Pascal Procedure Library* manual for details on the serial interface registers.

Using Compatibility Hardware

This method involves installing an HP 98546 Compatibility Video Card Set, which essentially contains the *separate* graphics and alpha planes of the Series 200 Model 217 computer. You can then direct the system to use the compatibility display, enabling you to run some existing Series 200 programs on your Series 300 computer. This section describes when and how to use this method of porting.

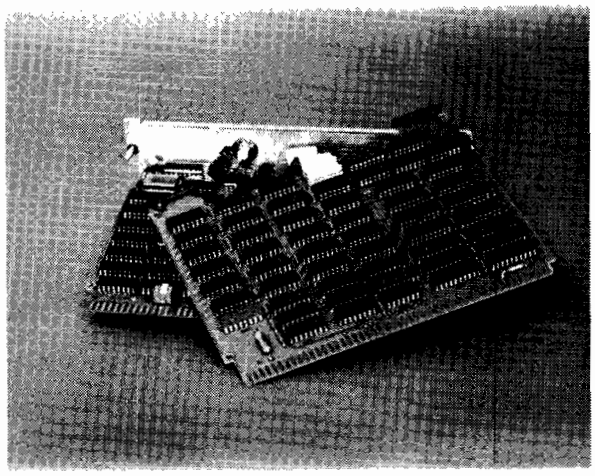
This card set remedies the following situations.

- The program depends on having separate alpha and graphics memory planes.
- The program directly accesses alpha or graphics hardware of a Model 217 or 236A computer (by writing directly into the screen's memory addresses, rather than through a higher-level Pascal or DGL procedure or function).
- The program depends on blinking or half-bright alpha display highlights (characters with codes 130, 131, and 134 through 143).
- The program depends on the Model 217's specific graphics resolution (512×390 pixels), alpha display size (80×25 characters), or on the registration of alpha and graphics pixels.

This method is required if any of the above statements is true **and** you cannot modify a program's source code (or don't want to). If you have the program's source code, then you may want to instead make the necessary changes in it.

Hardware Description

The card set consists of these two hardware pieces:

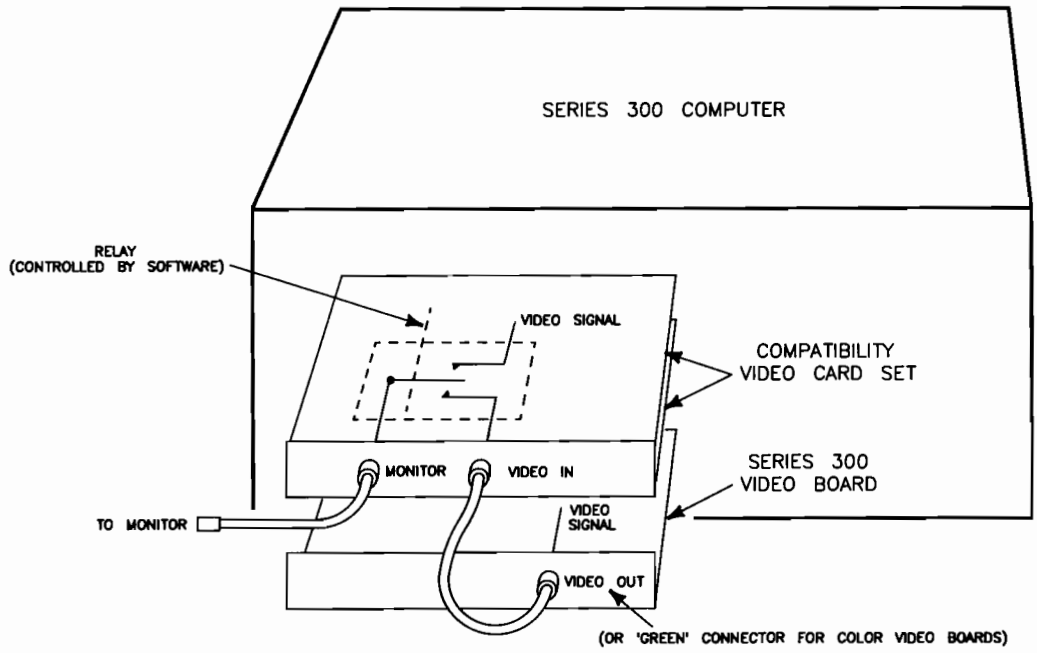


The Compatibility Video Card Set

- The alpha display card is like the existing 98204B display controller card, except for a relay and an additional BNC video connector on the rear panel.
- The graphics card which is identical to the Model 217's graphics card.

The Relay and BNC Video Connectors

The relay on the alpha card is used to switch between using the Series 300's display signal and using the compatibility display's signal.



A Relay Governs Which Display Signal Is Used

Compatibility Video Card Set Capabilities

Capabilities of this card are identical to those of the Model 217. The alpha display is an 80×25-character screen with half-bright, blinking, underline, and inverse-video display enhancements. The graphics display is 512×390 monochrome pixels.

Configurations Possible

Here are the video-interface/monitor configurations possible:

- **Shared monitor:** The Compatibility Video Card Set and the Series 300 Video Board can share a medium-resolution monitor (monochrome or color).
- **Separate monitors:** The Compatibility Video Card Set can use a medium-resolution monitor, and the Series 300 High-Resolution Video Board can use a separate high-resolution monitor (monochrome or color).
- **Single monitor:** The Compatibility Video Card Set can use a medium-resolution monitor (*with* no Series 300 video board or monitor).

Steps in Using this Card Set

Here are the steps you will take with this method:

1. Turn off the computer.
2. Configure and install the Compatibility Video Card Set according to the instructions in its *Installation Note*. Also connect the monitor(s) as described in that note.
3. Boot the system with the disc that uses the desired display hardware.
    a. If you want to use the Compatibility Video Card Set's display hardware, boot the Pascal system using the BOOT: disc. (It is similar to the BOOT: disc supplied with Pascal 3.0 and 3.01, as it contains the same driver modules as in the INITLIB file.)
    b. If you want to use the Series 300's bit-mapped display, boot the Pascal System using the BOOT2: disc. (The INITLIB file on BOOT2: contains the modules CRTC, CRTE, and CRTD, which are the alpha drivers for the Series 300 bit-mapped displays and the 98700 Display Controller. It does not contain the CRT, CRTB, CHOOK, or BAT modules required for Series 200.)

Note

Since you are using one monitor for two different displays, a small amount of time is required for the monitor to synchronize with the new display whenever you switch from one display to the other. This will sometimes cause the screen to flicker at power-up or after a soft re-boot. This normally occurs after the `Loading 'INITLIB'` message but before the `Loading 'STARTUP'` message appears on the screen.

Modifying the Source Program

This method involves possibly modifying the program's source code, possibly re-compiling using the 3.2 compiler, and possibly re-linking using the 3.2 libraries. This section describes when and how to use this method of porting programs.

This method is required for the following situations:

- Programs compiled on the 2.1 or earlier versions of the system.
- The program's object[1] file contains a *linked-in* 3.0 (or later) module that is *incompatible* with either the 3.2 system or Series 300 hardware (such as Device-independent Graphics, DGL, modules, heap manager HPM).
- The program uses any procedures below the level of Workstation Pascal or Procedure Library features (such as the "clock" procedures described in the "System Devices" chapter of the *Pascal Procedure Library* manual).
- The program uses HP 98203 [EDIT] or [RUN] key codes, which cannot be generated by the HP-HIL (HP 46020/21) keyboard.
- You want to fully utilize Series 300 hardware features which were not present on Series 200 computers (such as use features of the MC68020 processor or MC68881 co-processor).
- The program depends on an ID PROM (this is a memory location that permanently stores the computer's serial number).
- Your program treats pointers as 24-bit instead of 32-bit values as they are on the Model 330, 332, 340, 350, 360 and 370 computers.

If **any** of the above statements is **true**, then you probably need to modify and re-compile the program on the 3.2 system. If you do **not** have access to the source code (or separate object module in the case of the linked modules), then you **cannot** port it — you will have to buy a Series 300 version of the program, if it is available.

Programs Compiled on Pascal 2.1 (or Earlier Versions)

If your program was compiled on the Pascal 2.1 system (or an earlier version), then it will not run on the 3.2 system. You will have to re-compile the source code on the 3.2 system.

If your "pre-3.0" program uses any of the "internal" operating system modules (such as KBD or BAT), you will probably need to re-write the corresponding section of code since these operating system modules were re-designed with the 3.0 system. See the "System Devices" chapter of the *Pascal Procedure Library* manual for details on the new SYSDEVS operating system module.

[1] In this situation, you may not need to modify the source program. You may only need the program's separate *object* file (i.e., the program without the modules linked to it).

HP 98203 Specific Key Codes

The 98203 keyboards can generate [EDIT] and [RUN] key codes which cannot be generated by a 46020/21 keyboard. If your program depends on trapping these key codes, you will need to modify it to use 46020/21 keys instead. See the "System Devices" chapter of the *Pascal Procedure Library* manual for examples of trapping keystrokes with a Pascal program.

Linked-In, Incompatible Modules

An example of this situation is a program that used 3.0 DGL (Device-independent Graphics) module(s), and the required module(s) are linked to the program (i.e., the modules have been put into the program's object file and linked to it using the Librarian's Link command). Even though you may try to make the program use the 3.2 DGL modules by P-loading them and then running the program, the program will still access the linked-in 3.0 modules. Neither can you remove the linked-in 3.0 modules, since you cannot separate modules in an object file once they have been linked.

To remedy this situation, you will need to have the program's object code and use the Librarian to re-link to it the corresponding 3.2 module(s) that it requires. Note that the pre-3.2 heap manager (module HPM in SYSVOL:LIBRARY) will not work with the Models 330 and 350. The heap manager is used whenever the source program was compiled with **`$heap_dispose on$`**, and uses **`new`** or **`dispose`** on a pointer.

Use of Low-Level Procedures

If your program uses any low-level operating system modules, such as SYSDEVS for clock access, then you should probably re-compile it. The reason for this recommendation is that the interface text of these modules may have been modified slightly (and the system does not report any warning message for this type of situation). Source changes are probably required.

Full Utilization of Series 300 Hardware Features

An example of this situation is that programs compiled on a 3.0 or earlier system will not make use of MC68881/MC68882 floating-point math co-processor available on some Series 300 computers.

You can re-compile the program with the COMPILE20 compiler, and the program will make use of this hardware (if installed). Code compiled with COMPILE20 will fail if run on a Model 310 or a Series 200 computer.

The Knob on the HP-HIL 98203C keyboard requires the **`HPHIL`** and **`MOUSE`** modules in order to generate arrow keys.

Pascal Compiler Syntax Errors

ANSI/ISO Pascal Errors

1 Erroneous declaration of simple type.
2 Expected an identifier.
4 Expected a right parenthesis ")".
5 Expected a colon ":".
6 Symbol is not valid in this context.
7 Error in parameter list.
8 Expected the keyword OF.
9 Expected a left parenthesis "(".
10 Erroneous type declaration.
11 Expected a left bracket "[".
12 Expected a right bracket "]".
13 Expected the keyword END.
14 Expected a semicolon ";".
15 Expected an integer.
16 Expected an equal sign "=".
17 Expected the keyword BEGIN.
18 Expected a digit following '.'.
19 Error in field list of a record declaration.
20 Expected a comma ",".
21 Expected a period ".".
22 Expected a range specification symbol "..".
23 Expected an end-of-comment delimiter.
24 Expected a dollar sign "$".
50 Error in constant specification.
51 Expected an assignment operator ":=".
52 Expected the keyword THEN.
53 Expected the keyword UNTIL.
54 Expected the keyword DO.
55 Expected the keyword TO or DOWNTO.
56 Variable expected.
58 Erroneous factor in expression.
59 Erroneous symbol following a variable.
98 Illegal character in source text.
99 End of source text reached before end of program.
100 End of program reached before end of source text.
101 Identifier was already declared.
102 Low bound greater than high bound in range of constants.
103 Identifier is not of the appropriate class.
104 Identifier was not declared.
105 Non-numeric expressions cannot be signed.
106 Expected a numeric constant here.
107 Endpoint values of range must be compatible and ordinal.
108 NIL may not be redeclared.
110 Tagfield type in a variant record is not ordinal.
111 Variant case label is not compatible with tagfield.
113 Array dimension type is not ordinal.
115 Set base type is not ordinal.
117 An unsatisfied forward reference remains.
121 Pass by value parameter cannot be type FILE.
123 Type of function result is missing from declaration.
125 Erroneous type of argument for built-in routine.
126 Number of arguments different from number of formal parameters.
127 Argument is not compatible with corresponding parameter.
129 Operands in expression are not compatible.
130 Second operand of IN is not a set.
131 Only equality tests (= and < >) allowed on this type.
132 Tests for strict inclusion (< or >) not allowed on sets.
133 Relational comparison not allowed on this type.
134 Operand(s) are not proper type for this operation.
135 Expression does not evaluate to a boolean result.
136 Set elements are not of ordinal type.
137 Set elements are not compatible with set base type.
138 Variable is not an ARRAY structure.
139 Array index is not compatible with declared subscript.
140 Variable is not a RECORD structure.
141 Variable is not a pointer or FILE structure.
142 Packing allowed only on last dimension of conformant array.
143 FOR loop control variable is not of ordinal type.
144 CASE selector is not of ordinal type.
145 Limit values not compatible with loop control variable.
147 Case label is not compatible with selector.
149 Array dimension is not bounded.
150 Illegal to assign value to built-in function identifier.
152 No field of that name in the pertinent record.
154 Illegal argument to match pass-by-reference parameter.
156 Case label has already been used.
158 Structure is not a variant record.
160 Previous declaration was not FORWARD.
163 Statement label not in range 0..9999.
164 Target of nonlocal GOTO not in outermost compound statement.
165 Statement label has already been used.
166 Statement label was already declared.
167 Statement label was not declared.
168 Undefined statement label.
169 Set base type is not bounded.
171 Parameter list conflicts with forward declaration.
177 Cannot assign value to function outside its body.
181 Function must contain assignment to function result.
182 Set element is not in range of set base type.
183 File has illegal element type.
184 File parameter must be of type TEXT.
185 Undeclared external file or no file parameter.
190 Attempt to use type identifier in its own declaration.
300 Division by zero.
301 Overflow in constant expression.
302 Index expression out of bounds.
303 Value out of range.
304 Element expression out of range.
400 Unable to open list file.
401 File or volume not found.
403-409 Compiler errors.

Compiler Options

600 Directive is not at beginning of the program.
601 Indentation too large for PAGEWIDTH.
602 Directive not valid in executable code.
604 Too many parameters to SEARCH.
605 Conditional compilation directives out of order.
606 Feature not in standard Pascal flagged by ANSI ON.
607 Feature only allowed when UCSD enabled.
608 INCLUDE exceeds maximum allowed depth of files.
609 Cannot access this INCLUDE file.
610 INCLUDE or IMPORT nesting too deep.
611 Error in accessing library file.
612 Language extension not enabled.
613 Imported module does not have interface text.
614 LINENUM must be in the range 0..65535.
620 Only first instance of routine may have ALIAS.
621 ALIAS not in procedure or function header.
646 Directive not allowed in EXPORT section.
647 Illegal file name.
648 Illegal operand in compiler directive.
649 Unrecognized compiler directive.

Implementation Restrictions

651 Reference to a standard routine that is not implemented.
652 Illegal assignment or CALL involving a standard procedure.
653 CONST, TYPE, VAR, or MODULE cannot follow routine.
655 Record or array constructor not allowed in executable statement.
657 Loop control variable must be local variable.
658 Sets are restricted to the ordinal range 0..8175 (default) or 0..261999 (max).
659 Cannot blank pad literal to more than 255 characters.
660 String constant cannot extend past text line.
661 Integer constant exceeds the range implemented.
662 Nesting level of identifier scopes exceeds maximum (20).
663 Nesting level of declared routines exceeds maximum (15).
665 CASE statement must have non-OTHERWISE clause.
667 Routine was already declared FORWARD.
668 FORWARD routine may not be EXTERNAL.
671 Procedure too long.
672 Structure is too large to be allocated.
673 File component size must be in range 1..32766.
674 Field in record constructor improper or missing.
676 Structured constant has been discarded (cf. SAVE_CONST).
677 Constant overflow.
678 Allowable string length is 1..255 characters.
679 Range of case labels too large.
680 Real constant has too many digits.
681 Real number not allowed.
682 Error in structured constant.
683 More than 32 767 bytes of data.
684 Expression too complex.
685 Variable in READ or WRITE list exceeds 32 767 bytes.
686 Field width parameter must be in range 0..255.
687 Cannot IMPORT module name in its EXPORT section.
688 Structured constant not allowed in FORWARD module.
689 Module name may not exceed 15 characters.
696 Array elements are not packed.
697 Array lower bound is too large.
698 File parameter required.
699 32-bit arithmetic overflow.

Non-ISO Language Features

701 Cannot dereference variable of type ANYPTR.
702 Cannot make an assignment to this type of variable.
704 Illegal use of module name.
705 Too many concrete modules.
706 Concrete or external instance required.
707 Variable is of type not allowed in variant records.
708 Integer following "#" is greater than 255.
709 Illegal character in a "#" string.
710 Illegal item in EXPORT section.
711 Expected the keyword IMPLEMENT.
712 Expected the keyword RECOVER.
714 Expected the keyword EXPORT.
715 Expected the keyword MODULE.
716 Structured constant has erroneous type.
717 Illegal item in IMPORT section.
718 CALL to other than a procedural variable.
719 Module already implemented (duplicate module).
720 Concrete module not allowed here.
730 Structured constant component incompatible with corresponding type.
731 Array constant has incorrect number of elements.
732 Length specification required.
733 Type identifier required.
750 Error in constant expression.
751 Function result type must be assignable.
900 Insufficient space to open code file.
901 Insufficient space to open REF file.
902 Insufficient space to open DEF file.
903 Error in opening code file.
904 Error in opening REF file.
905 Error in opening DEF file.
906 Code file full.
907 REF file full.
908 DEF file full.

I/O System Errors

These are the values found in the system variable IORESULT and the corresponding error message which the system prints out automatically for you.

0	No I/O error reported.
1	Parity (CRC) wrong. I/O driver will do several retries.
2	Illegal unit number – valid range is 1..50.
3	Illegal I/O request (e.g., read from printer).
4	Device timeout.
5	Volume went off-line.
6	File lost in directory.
7	Bad file name.
8	No room on volume.
9	Volume not found.
10	File not found.
11	Duplicate directory entry.
12	File already open.
13	File not open.
14	Bad input format.
15	Disc block out of range.
16	Device absent or inaccessible.
17	Media initialization failed.
18	Media is write-protected.
19	Unexpected interrupt.
20	Hardware/media failure.
21	Unrecognized error state.
22	DMA absent or unavailable.
23	File size not compatible with type.
24	File not opened for reading.
25	File not opened for writing.
26	File not opened for direct access.
27	No room in directory.
28	String subscript out of range.
29	Bad string parameter on close of file.
30	Attempt to read past end-of-file mark.
31	Media not initialized.
32	Block not found.
33	Device not ready or media absent.
34	Media absent.
35	No directory on volume.
36	File type illegal or does not match request.
37	Parameter illegal or out of range.
38	File cannot be extended.
39	Undefined operation for file.
40	File not lockable.
41	File already locked.
42	File not locked.
43	Directory not empty.
44	Too many files open on device.
45	Access to file not allowed.
46	Invalid password.
47	File is not a directory.
48	Operation not allowed on a directory.
49	Cannot create /WORKSTATIONS/TEMP_FILES.
50	Unrecognized SRM error.
51	Medium may have been changed.
52	File system corrupt.
53	File or file system too big.
54	No permission for requested action.
55	Driver cache full.
56	Driver configuration failed.
57	IORESULT was 57.

Graphics System Errors

When writing graphics programs, it will be helpful to enclose the main body of the program in a TRY block. In the RECOVER block, test the value of ESCAPECODE. If ESCAPECODE=-27, invoke a graphics function called GRAPHICSERROR. This will return a number which can be cross-referenced with the following list of error messages.

0	No errors since last call to GRAPHICSERROR or INIT_GRAPHICS.
1	Graphics system not initialized.
2	Graphics display is not enabled.
3	Locator device not enabled.
4	ECHO value requires a graphic display to be enabled.
5	Graphics system is already enabled.
6	Illegal aspect ratio specified.
7	Illegal parameters specified.
8	Parameters specified are outside physical display limits.
9	Parameters specified are outside limits of window.
10	Logical locator and logical display use same device.
11	Parameters specified are outside virtual coordinate system boundary.
12	Escape function requested not supported by display device.
13	Parameters specified are outside physical locator limits.

Loader/SEGMENTER Errors

Here is a list of errors that can be generated by the loader or by a program that uses the SEGMENTER module.

100..105	Field overflow trying to link or relocate something.
110	Circular or too deeply nested symbol definitions.
111	Improper link information format.
112	Not enough memory.
116	File was not a code file.
117	Not enough space in the explicit global area.
118	Incorrect version number.
-119/119	Unresolved external references.
120	Generated by the dummy procedure returned by FIND_PROC.
121	UNLOAD_SEGMENT called when there are no more segments to unload.
122	Not enough space in the explicit code area.

I/O Library Errors

These are the values and corresponding error messages that may develop when using the I/O library. A call to IOERROR_MESSAGE will generate the appropriate message.

0	No error.
1	No card at select code.
2	Interface should be HP-IB.
3	Not active controller/commands not supported.
4	Should be device address, not select code.
5	No space left in buffer.
6	No data left in buffer.
7	Improper transfer attempted.
8	The select code is busy.
9	The buffer is busy.
10	Improper transfer count.
11	Bad timeout value/timeout not supported.
12	No driver for this card.
13	No DMA.
14	Word operations not allowed.
15	Not addressed as talker/write not allowed.
16	Not addressed as listener/read not allowed.
17	A timeout has occurred/no device.
18	Not system controller.
19	Bad status or control.
20	Bad set/clear/test operation.
21	Interface card is dead.
22	End/eod has occurred.
23	Miscellaneous-value of parameter error.
306	Datacomm interface failure.
313	USART receive buffer overflow.
314	Receive buffer overflow.
315	Missing clock.
316	CTS false too long.
317	Lost carrier disconnect.
318	No activity disconnect.
319	Connection not established.
325	Bad data bits/parity combination.
326	Bad status/control register.
327	Control value out of range.

Operating System Runtime Error Messages

Errors detected by the operating system during the execution of a program generate one of the following error messages. The numbers correspond to the value of ESCAPECODE.

0	Normal termination.
-1	Abnormal termination.
-2	Not enough memory.
-3	Reference to NIL pointer.
-4	Integer overflow.
-5	Divide by zero.
-6	Real math overflow. The number was too large.
-7	Real math underflow. The number was too small.
-8	Value range error.
-9	Case value range error.
-10	Non-zero IORESULT. (See "I/O System Errors".)
-11	CPU word access to odd address.
-12	CPU bus error.
-13	Illegal CPU instruction.
-14	CPU privilege violation.
-15	Bad argument – SIN/COS.
-16	Bad argument – LN (natural log).
-17	Bad argument – SQRT (square root).
-18	Bad argument – real/BCD conversion.
-19	Bad argument – BCD/real conversion.
-20	Stopped by user.
-21	Unassigned CPU trap.
-22	Reserved.
-23	Reserved.
-24	Macro parameter not 0..9 or a..z.
-25	Undefined macro parameter.
-26	Non-zero IOE-RESULT. (See "I/O Library Errors".)
-27	Non-zero GRAPHICSERROR. (See "Graphics System Errors".)
-28	Parity error in memory.
-29	Miscellaneous hardware floating-point error.
-30	Bad argument – arcsine/arccosine. Argument >1.
-31	Illegal real number.

VMELIBRARY Errors

When a VME error occurs while using the VME_DRIVER module procedures, you can determine which has occurred by using a TRY...RECOVER construct and calling the ESCAPECODE function in the RECOVER block.

800	Range error: select code <7 or >31.
801	Tried to access the HP VMEbus Interface using an odd Select Code.
802	Timeout error, the VMEbus System Controller does not grant the bus to the HP VMEbus Interface within the amount of seconds specified in the last 'SET_TIMEOUT' call.
803	NumOfChar <0 or $>$ declared size of 'Data' in VME_StrRead NumOfBytes <0 VME_BlockRead or VME_BlockWrite.
805	Odd NumOfBytes when using Transfer mode WordInc or WordFxd.
806	The VMEbus Interface Card is not an HP 98646A VMEbus Interface Card.

Error Messages A

This appendix contains all of the error messages and conditions that you are likely to encounter while using the Pascal system. They can be placed into the following categories; each category is discussed in a subsequent section.

- Unreported errors — certain errors do not get reported by this implementation of Pascal.
- Boot-time errors — these are errors that occur while the Pascal system is booting (they are reported by the system loader).
- Run-time errors — These are general errors which may occur while you are using the system.

  Run-time errors −10, −26, and −27 have special meanings:

  - I/O System errors — When run-time error −10 occurs, there has been a problem with the I/O system. The operating system then prints an error message from the list of I/O system errors.
  - I/O Library errors — When run-time error −26 occurs, there has been a problem in an `IO` library procedure.
  - Graphics Library errors — When run-time error −27 occurs, there has been a problem in a `GRAPHICS` library procedure.
- Loader/SEGMENTER errors.
- Compiler syntax errors.
- Assembler errors and conditions.
- Debugger errors and conditions.

Unreported Errors

The following errors in Pascal programs are not reported by this implementation of the language.

- Disposing a pointer while in the scope of a **WITH** referencing the variable to which it points.
- Disposing a pointer while the variable it points to is being used as a **var** parameter.
- Disposing an uninitialized or NIL pointer.
- Disposing a pointer to a variant record using the wrong tagfield list.
- Assignment to a **FOR**-loop control variable while inside the loop.
- **GOTO** into a conditional structured statement.
- Exiting a function before a result value has been assigned.
- Changing the tagfield of a dynamic variable to a value other than what was specified in the call to **NEW**.
- Accessing a variant field when the tagfield indicates a different variant.
- Negative field width parameters in a **WRITE** statement.
- The underscore character "_" is allowed in identifiers. This is permitted in HP Pascal, but is not reported as an error when compiling with **$ANSI$** specified.
- Value range error is not always reported when an illegal value is assigned to a variable of type **SET**.

Boot-Time Errors

Errors that occur while your system is booting will report a message like this:

```
IORESULT, ERROR:   0, 112
```

The value of **IORESULT** is shown first (**0** in the above display). See the I/O System Errors section for descriptions of those error numbers.

The value of **ERROR** is shown second (**112** in the above display). See the Loader/SEGMENTER Errors section for a description of those error numbers.

Run-Time Errors

Errors detected by the operating system during the execution of a program generate one of the error messages listed on this page (unless you trap it with a TRY..RECOVER construct).

Note

Note that when error −10 occurs, the error message listed here will *not* be shown; the message on the next page (in I/O System Errors) will be shown instead.

When using a TRY..RECOVER construct (which requires the $SYSPROG ON$ Compiler option), the following numbers correspond to the value returned by the ESCAPECODE function.

0 Normal termination.
-1 Abnormal termination.
-2 Not enough memory.
-3 Reference to NIL pointer.
-4 Integer overflow.
-5 Divide by zero.
-6 Real math overflow. The number was too large.
-7 Real math underflow. The number was too small.
-8 Value range error.
-9 Case value range error.
-10 Non-zero IORESULT. (See "I/O System Errors".)
-11 CPU word access to odd address.
-12 CPU bus error.
-13 Illegal CPU instruction.
-14 CPU privilege violation.
-15 Bad argument – SIN/COS.
-16 Bad argument – LN (natural log).
-17 Bad argument – SQRT (square root).
-18 Bad argument – real/BCD conversion.
-19 Bad argument – BCD/real conversion.
-20 Stopped by user.
-21 Unassigned CPU trap.
-22 Reserved.
-23 Reserved.
-24 Macro parameter not 0..9 or a..z.
-25 Undefined macro parameter.
-26 Non-zero IOE-RESULT. (See "I/O Library Errors".)
-27 Non-zero GRAPHICSERROR. (See "Graphics System Errors".)
-28 Parity error in memory.
-29 Miscellaneous hardware floating-point error.
-30 Bad argument – arcsine/arccosine. Argument > 1.
-31 Illegal real number.

I/O System Errors

These error messages are automatically printed by the system unless you have enclosed the error-producing statement in a TRY..RECOVER construct. Within the RECOVER block, the ESCAPECODE function returning a value of −10 indicates that one of the following errors has occurred; you can determine which error has occurred by using the IORESULT function.

0 No I/O error reported.
1 Parity (CRC) wrong. I/O driver will do several retries.
2 Illegal unit number – valid range is 1..50.
3 Illegal I/O request (e.g., read from printer).
4 Device timeout.
5 Volume went off-line.
6 File lost in directory.
7 Bad file name.
8 No room on volume.
9 Volume not found.
10 File not found.
11 Duplicate directory entry.
12 File already open.
13 File not open.
14 Bad input format.
15 Disc block out of range.
16 Device absent or inaccessible.
17 Media initialization failed.
18 Media is write-protected.
19 Unexpected interrupt.
20 Hardware/media failure.
21 Unrecognized error state.
22 DMA absent or unavailable.
23 File size not compatible with type.
24 File not opened for reading.
25 File not opened for writing.
26 File not opened for direct access.
27 No room in directory.
28 String subscript out of range.
29 Bad string parameter on close of file.
30 Attempt to read past end-of-file mark.
31 Media not initialized.
32 Block not found.
33 Device not ready or media absent.
34 Media absent.
35 No directory on volume.
36 File type illegal or does not match request.
37 Parameter illegal or out of range.
38 File cannot be extended.
39 Undefined operation for file.
40 File not lockable.
41 File already locked.
42 File not locked.
43 Directory not empty.
44 Too many files open on device.
45 Access to file not allowed.
46 Invalid password.
47 File is not a directory.
48 Operation not allowed on a directory.
49 Cannot create /WORKSTATIONS/TEMP_FILES.

50 Unrecognized SRM error.
51 Medium may have been changed.
52 File system is corrupt.
53 File system or file is bigger than 2^{31} - 1 bytes.
54 No permission for requested access.
55 File system cache full.
56 Driver configuration failed.

I/O Library Errors

When run-time error −26 occurs, there has been a problem in an I/O library procedure. By importing the IODECLARATIONS module, you can use the IOE_RESULT and IOERROR_MESSAGE functions to get a textual error description. For example:

```
$SYSPROG ON$
      ...
import IODECLARATIONS, GENERAL_3
      ...
begin
try
      ...
recover
    if ESCAPECODE = IOESCAPECODE then writeln (IOERROR_MESSAGE(IOE_RESULT));
    ESCAPE(ESCAPECODE);
end.
```

IOESCAPECODE is a constant (= −26) which you can import from the IODECLARATIONS module. ESCAPE is a procedure and ESCAPECODE is a function; both are accessible when you use the $SYSPROG ON$ Compiler option.

0 No error.
1 No card at select code.
2 Interface should be HP-IB.
3 Not active controller/commands not supported.
4 Should be device address, not select code.
5 No space left in buffer.
6 No data left in buffer.
7 Improper transfer attempted.
8 The select code is busy.
9 The buffer is busy.
10 Improper transfer count.
11 Bad timeout value/timeout not supported.
12 No driver for this card.
13 No DMA.
14 Word operations not allowed.
15 Not addressed as talker/write not allowed.
16 Not addressed as listener/read not allowed.
17 A timeout has occurred/no device.
18 Not system controller.
19 Bad status or control.
20 Bad set/clear/test operation.
21 Interface card is dead.
22 End/eod has occurred.
23 Miscellaneous-value of parameter error.
306 Datacomm interface failure.
313 USART receive buffer overflow.
314 Receive buffer overflow.
315 Missing clock.
316 CTS false too long.
317 Lost carrier disconnect.
318 No activity disconnect.
319 Connection not established.
325 Bad data bits/parity combination.
326 Bad status/control register.
327 Control value out of range.

Graphics Errors

When run-time error −27 occurs, there has been an error in a GRAPHICS library routine.

By importing the DGL_LIB module, you can call the GRAPHICSERROR function which returns an INTEGER value you can cross reference with the numbered list of graphics errors.

```
$SYSPROG ON$
      ...
import DGL_LIB;
      ...
begin
try
      ...
recover
    if ESCAPECODE = -27
        then writeln ('Graphics error #', GRAPHICSERROR,' has occurred')
        else ESCAPE(ESCAPECODE);
end.
```

You may wish to write a procedure which takes the INTEGER value from GRAPHICSERROR and prints the description of the error on the CRT. You could keep this procedure with your program or, for more global use, in the System Library (normally SYSVOL:LIBRARY).

0 No errors since last call to GRAPHICSERROR or INIT_GRAPHICS.

1 Graphics system not initialized.

2 Graphics display is not enabled.

3 Locator device not enabled.

4 ECHO value requires a graphic display to be enabled.

5 Graphics system is already enabled.

6 Illegal aspect ratio specified.

7 Illegal parameters specified.

8 Parameters specified are outside physical display limits.

9 Parameters specified are outside limits of window.

10 Logical locator and logical display use same device.

11 Parameters specified are outside virtual coordinate system boundary.

12 Escape function requested not supported by display device.

13 Parameters specified are outside physical locator limits.

Loader/SEGMENTER Errors

Here is a list of errors that can be generated by a program that uses the SEGMENTER module (or by the loader; see Boot-Time Errors):

100..105 Field overflow trying to link or relocate something.

110 Circular or too deeply nested symbol definitions.

111 Improper link information format.

112 Not enough memory.

116 File was not a code file.

117 Not enough space in the explicit global area.

118 Incorrect version number.

-119/119 Unresolved external references.

120 Generated by the dummy procedure returned by `find_proc`.

121 `unload_segment` called when there are no more segments to unload.

122 Not enough space in the explicit code area.

SEGMENTER Errors

When one of these errors occurs while using the SEGMENTER module procedures, you can determine which has occurred by using a TRY..RECOVER construct and calling the ESCAPECODE function in the RECOVER block.

Loader Boot-Time Errors

When an error occurs while booting, a message such as the following will be reported:

```
IORESULT, ERROR =     0,  112
```

The second number indicates which loader error has occurred. (The first number indicates which I/O system error has occurred; see the preceding I/O System Errors section for descriptions of each error.)

Pascal Compiler Errors

The following errors may occur during the compilation of a HP Pascal program.

1 Erroneous declaration of simple type.
2 Expected an identifier.
4 Expected a right parenthesis ")".
5 Expected a colon ":".
6 Symbol is not valid in this context.
7 Error in parameter list.
8 Expected the keyword OF.
9 Expected a left parenthesis "(".
10 Erroneous type declaration.
11 Expected a left bracket "[".
12 Expected a right bracket "]".
13 Expected the keyword END.
14 Expected a semicolon ";".
15 Expected an integer.
16 Expected an equal sign "=".
17 Expected the keyword BEGIN.
18 Expected a digit following '.'.
19 Error in field list of a record declaration.
20 Expected a comma ",".
21 Expected a period ".".
22 Expected a range specification symbol "..".
23 Expected an end-of-comment delimiter.
24 Expected a dollar sign "$".
50 Error in constant specification.
51 Expected an assignment operator ":=".
52 Expected the keyword THEN.
53 Expected the keyword UNTIL.
54 Expected the keyword DO.
55 Expected the keyword TO or DOWNTO.
56 Variable expected.
58 Erroneous factor in expression.
59 Erroneous symbol following a variable.
98 Illegal character in source text.
99 End of source text reached before end of program.
100 End of program reached before end of source text.
101 Identifier was already declared.
102 Low bound greater than high bound in range of constants.
103 Identifier is not of the appropriate class.
104 Identifier was not declared.
105 Non-numeric expressions cannot be signed.
106 Expected a numeric constant here.
107 Endpoint values of range must be compatible and ordinal.
108 NIL may not be redeclared.
110 Tagfield type in a variant record is not ordinal.
111 Variant case label is not compatible with tagfield.
113 Array dimension type is not ordinal.
115 Set base type is not ordinal.
117 An unsatisfied forward reference remains.
121 Pass by value parameter cannot be type FILE.
123 Type of function result is missing from declaration.
125 Erroneous type of argument for built-in routine.
126 Number of arguments different from number of formal parameters.
127 Argument is not compatible with corresponding parameter.
129 Operands in expression are not compatible.

130 Second operand of IN is not a set.
131 Only equality tests (= and < >) allowed on this type.
132 Tests for strict inclusion (< or >) not allowed on sets.
133 Relational comparison not allowed on this type.
134 Operand(s) are not proper type for this operation.
135 Expression does not evaluate to a boolean result.
136 Set elements are not of ordinal type.
137 Set elements are not compatible with set base type.
138 Variable is not an ARRAY structure.
139 Array index is not compatible with declared subscript.
140 Variable is not a RECORD structure.
141 Variable is not a pointer or FILE structure.
142 Packing allowed only on last dimension of conformant array.
143 FOR loop control variable is not of ordinal type.
144 CASE selector is not of ordinal type.
145 Limit values not compatible with loop control variable.
147 Case label is not compatible with selector.
149 Array dimension is not bounded.
150 Illegal to assign value to built-in function identifier.
152 No field of that name in the pertinent record.
154 Illegal argument to match pass-by-reference parameter.
156 Case label has already been used.
158 Structure is not a variant record.
160 Previous declaration was not FORWARD.
163 Statement label not in range 0..9999.
164 Target of nonlocal GOTO not in outermost compound statement.
165 Statement label has already been used.
166 Statement label was already declared.
167 Statement label was not declared.
168 Undefined statement label.
169 Set base type is not bounded.
171 Parameter list conflicts with forward declaration.
177 Cannot assign value to function outside its body.
181 Function must contain assignment to function result.
182 Set element is not in range of set base type.
183 File has illegal element type.
184 File parameter must be of type TEXT.
185 Undeclared external file or no file parameter.
190 Attempt to use type identifier in its own declaration.
300 Division by zero.
301 Overflow in constant expression.
302 Index expression out of bounds.
303 Value out of range.
304 Element expression out of range.
400 Unable to open list file.
401 File or volume not found.
403 – **409** Compiler errors.

Compiler Options

600 Directive is not at beginning of the program.
601 Indentation too large for PAGEWIDTH.
602 Directive not valid in executable code.
604 Too many parameters to SEARCH.
605 Conditional compilation directives out of order.
606 Feature not in standard Pascal flagged by ANSI ON.
607 Feature only allowed when UCSD enabled.
608 INCLUDE exceeds maximum allowed depth of files.
609 Cannot access this INCLUDE file.
610 INCLUDE or IMPORT nesting too deep.
611 Error in accessing library file.
612 Language extension not enabled.
613 Imported module does not have interface text.
614 LINENUM must be in the range 0..65535.
620 Only first instance of routine may have ALIAS.
621 ALIAS not in procedure or function header.
646 Directive not allowed in EXPORT section.
647 Illegal file name.
648 Illegal operand in compiler directive.
649 Unrecognized compiler directive.

Implementation Restrictions

651 Reference to a standard routine that is not implemented.
652 Illegal assignment or CALL involving a standard procedure.
653 CONST, TYPE, VAR, or MODULE cannot follow routine.
655 Record or array constructor not allowed in executable statement.
657 Loop control variable must be local variable.
658 Sets are restricted to the ordinal range 0..8175 (default) or 0..261999 (max).
659 Cannot blank pad literal to more than 255 characters.
660 String constant cannot extend past text line.
661 Integer constant exceeds the range implemented.
662 Nesting level of identifier scopes exceeds maximum (20).
663 Nesting level of declared routines exceeds maximum (15).
665 CASE statement must have non-OTHERWISE clause.
667 Routine was already declared FORWARD.
668 FORWARD routine may not be EXTERNAL.
671 Procedure too long.
672 Structure is too large to be allocated.
673 File component size must be in range 1..32766.
674 Field in record constructor improper or missing.
676 Structured constant has been discarded (cf. SAVE_CONST).
677 Constant overflow.
678 Allowable string length is 1..255 characters.
679 Range of case labels too large.
680 Real constant has too many digits.
681 Real number not allowed.
682 Error in structured constant.
683 More than 32767 bytes of data.
684 Expression too complex.
685 Variable in READ or WRITE list exceeds 32767 bytes.
686 Field width parameter must be in range 0..255.
687 Cannot IMPORT module name in its EXPORT section.
688 Structured constant not allowed in FORWARD module.
689 Module name may not exceed 15 characters.
696 Array elements are not packed.

697 Array lower bound is too large.
698 File parameter required.
699 32-bit arithmetic overflow.

Non-ISO Language Features

701 Cannot dereference variable of type ANYPTR.
702 Cannot make an assignment to this type of variable.
704 Illegal use of module name.
705 Too many concrete modules.
706 Concrete or external instance required.
707 Variable is of type not allowed in variant records.
708 Integer following "#" is greater than 255.
709 Illegal character in a "#" string.
710 Illegal item in EXPORT section.
711 Expected the keyword IMPLEMENT.
712 Expected the keyword RECOVER.
714 Expected the keyword EXPORT.
715 Expected the keyword MODULE.
716 Structured constant has erroneous type.
717 Illegal item in IMPORT section.
718 CALL to other than a procedural variable.
719 Module already implemented (duplicate module).
720 Concrete module not allowed here.
730 Structured constant component incompatible with corresponding type.
731 Array constant has incorrect number of elements.
732 Length specification required.
733 Type identifier required.
750 Error in constant expression.
751 Function result type must be assignable.
900 Insufficient space to open code file.
901 Insufficient space to open REF file.
902 Insufficient space to open DEF file.
903 Error in opening code file.
904 Error in opening REF file.
905 Error in opening DEF file.
906 Code file full.
907 REF file full.
908 DEF file full.

Assembler Errors

Error messages are listed under the line in which they occur. At the completion of the assembly, the number of errors will be displayed. If there are errors, there will be a directive for you to check the location of the last error in the program. At that location there will be a description of the error. Also listed will be the location of the error above it if one exists. In this manner, all errors can be located without having to search the whole listing.

Error Messages

Address Register Expected.
Attempt to Nest Included Files.
Blank or EOL Expected.
Comma Expected.
Code Segment Starts at Odd Address.
Duplicate Definition of Symbol.
Error Reading Source File.
Error Reading Code File.
Error Writing Source File.
Error Writing Code File.
Expression is Improper Mode.
External Reference Not Allowed.
Failed to Open Include File.
File could not be found.

Field Overflow
A specification of the assembly instruction will not fit within the appropriate field of the machine instruction.

Illegal Constant.
Illegal Expression.
Illegal Operand Size for this Instruction.
Illegal Syntax.
Improper Addressing Mode.
Improper Use of Mode Declaration.
Symbol already has mode or declaration appears after first use of symbol.

Debugger Error Messages/Conditions

ADDRESS ERROR
An odd address has been referenced when an even address is required.

ADDRESS FORMAT NOT ALLOWED
The *, <, >, and ^ format codes are allowed only if the object is type address.

BAD DIGIT
There is an invalid digit in a number, for instance 8 in an octal number, in the current command.

BAD SYSTEM NAME
In an SB command, the system name parameter is invalid.

BUSERROR
An address has been accessed which does not exist in the machine's configuration.

DIVIDE BY ZERO
The value to the right of the / symbol is zero.

DUPLICATE BREAK
GT or TT has specified a location which already has a break point defined.

EXPRESSION TOO COMPLEX
The expression requires too much stack space to execute; for example, having more than three levels of parentheses.

FORMAT REQUIRES MORE DATA
An attempt has been made to display more bytes than the object contains.

INPUT OVERFLOW
An internal input stack has overflowed.

... IS UNDEFINED SYMBOL
An expression contains a reference to a symbol which the debugger does not recognize.

MORE
For a Q command, there is more data to be displayed. Press Return or Enter to view that data.

NEXT PROC
The current PN command has completed, and a new procedure has just started.

NO STATIC LINK
A WS command was given, but there is no STATIC link in the current stack frame.

NOW AT LINE ...
A line specified in an active break point has been encountered. The debugger is now waiting for input.

NOW AT START
A program was started with the `D` command. The debugger now has control and is ready to execute the first instruction of the program.

OVERFLOW
A number entered or the result of an arithmetic operation cannot be represented in 32 bits.

PC NOW AT...
The instuction at the address specified in an active break point has been encountered. The debugger is now waiting for input.

PC/SP HAS ODD ADDRESS
An attempt to return to the user code has been made under the above conditions.

PROC EXITED
The current `PN` or `PX` command has completed, and the procedure executing when the command was given has exited.

RAM PARITY ERROR
A parity error in the system's main memory has been detected. The last operation may have been aborted or incorrectly done.

SIZE ERROR
An entered value does not fit in a required space such as a register.

SIZE FIELD TOO BIG
In a format, the size field is too large for the object being dumped or the format specification being used. The size field for I and U is 1..4. The default size for string data is the length of the string.

STATION ADDRESS ERROR
In an `sb` command, the LANID (STATION ADDRESS) parameter was syntactically invalid.

SYNTAX ERROR
The syntax rules for the current command have been violated.

TOO MANY CODES
There are too many escape codes in the ET or ETN list.

TYPE ERROR
The parameter entered for a command is not the correct type; for example, using an alpha value when a line number or address is required.

UNIT NUMBER INVALID FOR BOOT
In an `sb` comman, the MSUS parameter was coded as a unit number. That number references a nonexistent device or a device from which you cannot boot such as a CRT.

USER TRAP 15 AT ...
A TRAP 15 instruction has been encountered which was not placed in the code by the debugger. The debugger is now waiting for input.

WHAT?
The first characters of a command are not recognized.

VMELIBRARY Errors

When a VME error occurs while using the VME_DRIVER module procedures, you can determine which has occurred by using a TRY..RECOVER construct and calling the ESCAPECODE function in the RECOVER block.

800 Range Error: Select Code <7 or > 31.

801 Tried to access the HP VMEbus Interface using an odd Select Code

802 Timeout error, the VMEbus System Controller does not grant the bus to the HP VMEbus Interface within the amount of seconds specified in the last `'SET_TIMEOUT'` call.

803 NumOfChar <0 or > declared size of `Data` in VME_StrRead.
NumOfBytes < 0 in VME_BlockRead or VME-BlockWrite.

805 Odd NumOfBytes when using Transfer_mode WordInc or WordFxd.

806 The VMEbus Interface Card is not an HP98646A VMEbus Interface Card.

Technical Reference B

This appendix contains the following useful reference information.

- A "System History" section that describes the additional features provided by Pascal System versions 2.x and 3.x
- A discussion of file interchange between the Pascal System and Series 200/300 BASIC Systems
- A list of module names used by this Operating System
- A physical memory map
- A software memory map

System History

This section first briefly describes the 1.0 version Pascal Workstation System, and then describes each subsequent version from the standpoint of what features have been added or changed by the version. It is intended to help you make the transition from earlier versions of the system to the 3.0 system.

Pascal 1.0

Here is a brief description of the Pascal 1.0 System. It is put here in order to give you a reference point from which to begin the comparison of later systems.

System Discs

The Pascal 1.0 Workstation System was distributed on a set of four mini-floppy discs, plus one additional disc for documentation. Here are the disc names:

BOOT: SYSVOL: ACCESS: COMPASM: DOC:

The SYSVOL:SYSTEM.LIBRARY file contained the entire complement of IO, GRAPHICS, and INTERFACE modules. The unmodified BOOT:SYSTEM.INITLIB contained device-driver software for all peripheral devices supported by the 1.0 system.

Documentation

Documentation for the 1.0 system included the following five manuals.

Problem Solving and Programming with Pascal — This is the textbook from which you can learn about Pascal programming, if you don't already know how to program in this language.

Pascal Language System User's Manual — This manual described booting the system and using each of the subsystems, such as the Editor, Compiler, and Assembler.

Pascal Procedure Library User's Manual — This manual described using the libraries supplied with the system. The libraries consisted of I/O, graphics, LIF-ASCII Filer, Heap Management, and other procedures (etc.) provided with the system.

MC 68000 User's Manual — This manual described MC68000 processor hardware and instruction set.

The Pascal Handbook — This manual described the Pascal language and extensions supported by the Series 200 Computers.

Computers Supported by 1.0

Pascal 1.0 supported only the 9826 and 9836, since they were the only Series 200 computers in production at the introduction of the Pascal 1.0 Workstation System.

Peripheral Devices Supported by 1.0

Pascal 1.0 supported the following mass storage devices:

- Internal 5.25-inch flexible disc drive
- HP 9885 and 9895 8-inch Flexible Disc Drives
- HP 9134 Hard Disc Drives

Pascal 2.0 and 2.1

Here are the additions to the 1.0 system and differences between the 2.1, 2.0, and 1.0 versions of the system.

System Discs

Pascal 2.0 and 2.1 Systems were distributed on six system discs, plus one documentation disc. Here are the names of the discs.

BOOT:	SYSVOL:	ACCESS:	CMPASM:	LIB:
CONFIG:	DOC:			

In contrast to the 1.0 file, the 2.x SYSVOL:LIBRARY was almost empty; the IO, GRAPHICS, and INTERFACE libraries were supplied on separate discs. The user could put just the ones he wanted into his System Library (usually the LIBRARY file). In addition, the Pascal 2.1 GRAPHICS library was re-structured internally and at the user-procedure level.

The Initialization Library (BOOT:INITLIB) supplied contained device-driver software for the most common peripherals but not for all; this was done to conserve memory for the average user, since Pascal 2.x supported many more peripheral devices. The less commonly needed drivers were supplied on the separate CONFIG: disc. Thus, to configure a system to use certain peripherals, the Librarian needed to be used to install the required driver software in INITLIB. Documentation was provided which explained how and when to add optional modules to the INITLIB file.

Documentation

Documentation for the 2.0 system consisted of the five manuals supplied with the 1.0 system, plus the additional *System Internals Documentation* set. This set consisted of these three manuals:

Pascal 2.0 System Designer's Guide — This manual described much of the inner workings of the Pascal system. It contained enough detail to allow you to use many of the "kernel" modules, and it also provided a fairly detailed description of Boot ROM contents and internal computer (hardware and software) architecture.

Pascal 2.0 Source Code Listings (Volume I) — This manual consisted of a cross reference of Pascal procedure names used in the system, and listings of Assembler language modules in the system.

Pascal 2.0 Source Code Listings (Volume II) — This manual consisted of the listings of many Pascal modules used in the system.

File System

HP's Logical Interchange Format (LIF) directory structure was made the primary disc organization for 2.0 and later versions. (LIF ASCII files are intended for interchangeability with other HP products.) The 1.0 file system was only able to cleanly handle UCSD directory organizations. HP provided a library of routines to access LIF discs, but they were not integrated into the File System.

The LIF library is not present in the 2.0 and later versions, since it is no longer necessary. The Lfiler (LIF Filer) is also unnecessary and has gone away, since the standard system Filer can now do the job. The 2.0 and later Filers are completely revised programs, although their behaviors are as similar as possible to the 1.0 Filer.

If you were using the 1.0 version and are switching to a later release, don't panic! This does not mean that Pascal 1.0 discs are inaccessible, or even that you need to convert them. See the Special Configurations section of the Technical Reference Appendix for details.

The 2.0 and later File Systems are completely reorganized in comparison to the 1.0 File System. The File System is now broken into levels called File Support (FS), Directory Access Method (DAM), Access Method (AM), and Transfer Method (TM). This organization allows the system to handle any number of different directory formats, and separates out the processing of each type of file structure which is supported. In fact, a customer can invent a new directory format or file type and bind it into the system so it can be used by all programs.

The Directory Access Methods supported in revisions 2.0 and 2.1 are as follows:

- HP Logical Interchange Format (LIF)
- Shared Resource Manager hierarchical "structured" format (SDF)
- UCSD-compatible (same format as Pascal 1.0)

All these directory organizations are available through normal Pascal file operations. Files generated under Pascal 1.0 are all still fully usable. However, the newer systems can generate files and discs which cannot be properly interpreted by the 1.0 File System.

System File Names

The names of system files were changed with the 2.0 system. They were changed because their length was longer than allowed by the LIF directory format. The name changes are as follows:

Old 1.0 File Name	New Name
SYSTEM.LINKER	LIBRARIAN
SYSTEM.EDITOR	EDITOR
SYSTEM.FILER	FILER
SYSTEM.COMPILER	COMPILER
SYSTEM.ASSMBLER	ASSEMBLER
SYSTEM.LIBRARY	LIBRARY
SYSTEM.TABLE	TABLE
SYSTEM.INITLIB	INITLIB
SYSTEM.MISCINFO	MISCINFO
SYSTEM.STARTUP	STARTUP

Object Code Compatibility

Several internal File System changes were made with Pascal 2.0. These changes resulted in corresponding changes in the internal representation of object code files. In general, when a version of the system is not compatible with other versions, the leading digit of the version number will be changed. For instance, versions 2.0 and 1.0 are not object-code compatible, while versions 2.1 and 2.0 are.

While it is regrettable, there really is no alternative to these compatibility restrictions. On the positive side, Pascal application programs which don't "fiddle around" in the operating system are forward compatible to 2.x, so recompilation is all that's necessary.

Supervisor Vs. User State

In versions 2.0 and later, user programs run in the 68000's "user" privilege mode, using the user stack pointer (USP). Interrupts run in "supervisor" privilege mode, using the system stack pointer (SSP). This has implications for calling Boot ROM routines, etc. See the *MC60000 User's Manual* for further details regarding these states.

Additional Computer Supported by 2.0

- Model 16 (HP 9816)

Additional Computer Supported by 2.1

- Model 20 (HP 9920)

Peripheral Configuration

The Pascal 2.x BOOT:TABLE auto-configuration program scanned interfaces for various peripherals and automatically assigned File System unit numbers to devices found (if possible). That was a considerable improvement over the 1.0 version of TABLE.

A source-code version (CTABLE.TEXT) was provided with the system. You could look at the program and read the corresponding commentary in the Special Configurations section of the Technical Reference Appendix to see exactly how the auto-configuration program works. You could also modify certain portions of it to make your own special configurations.

Additional Peripherals Supported by 2.0

Here are changes to the list of disc peripherals supported by Pascal 2.0.

- The CS/80 discs (7908 family)
- The Shared Resource Management system
- The HP 8920x 5.25-inch Flexible Disc Drives
- The HP 9121 3.5-inch (Single-Sided) Flexible Disc Drives
- Several new versions of the HP 913x Hard Disc Drives (they appear as one large volume instead of four smaller ones)
- Certain less obvious features were also added. For instance, the 2.0 system could be fairly easily configured to run from a terminal instead of the built-in CRT and keyboard.

Miscellaneous

Up to 65K bytes of Global space has been made available with 2.0 and later versions. This change involved a redefinition of the use of register A5, which now points to an address 32K bytes *below* the start of Globals rather than above the first global variable. Consequently, routines in the Boot ROM cannot any longer be called directly; a small interfacing routine is now required to set up the registers and fool the TRY-RECOVER mechanism when calling Boot ROM routines.

Pascal 3.0

Here are the differences and additional features provided by the 3.0 version of the system.

System Discs

The Pascal 3.0 System is distributed on 8 discs, plus two for documentation. Here are the names of the discs.

BOOT:	SYSVOL:	ACCESS:	CMP:	ASM:
LIB:	FLTLIB:	CONFIG:	DOC:	DGLPRG:

The BOOT:INITLIB file contains a more complete set of device-driver modules; for instance, it now contains module CS80 so that these discs will be recognized by the standard system. See the "Adding Modules to INITLIB" section of the "Special Configurations" chapter for a complete list of modules and descriptions of each.

Note that the Assembler and Compiler were put on separate discs due to size. The CMP: disc contains the Compiler. The ASM: disc contains the DEBUGGER program (formerly in BOOT:INITLIB) and the new REVASM (reverse assembler) module.

There are two versions of the GRAPHICS library. The FLTLIB:FGRAPHICS library contains modules optimized for using the HP 98635 Floating-Point Math card; they were compiled with the $FLOAT_HDW ON$ Compiler option, and use the 98635 card, if present. The LIB:GRAPHICS library uses routines in the REALS operating system module; these routines also access the Floating-Point Math card, if present, but the overhead in calling the routines decreases execution speed. The 98635 card can be used with all Pascal 3.0 programs, as long as the REALS module is installed (via INITLIB, etc.).

The DGLPRG: disc provides magnetic copy of the example programs given in the new *Pascal 3.0 Graphics Techniques* manual.

Documentation

Here are the documents shipped with the Pascal 3.0 system.

Pascal 3.0 User's Guide — This is a new manual that takes you from booting your system through setting up your "environment." It provides a "guided tour" of several subsystems, such as the Editor and Filer. You will see all of the steps required to enter, store, compile, and run a simple Pascal program.

Problem Solving and Programming with Pascal — This is the textbook from which you can learn about Pascal programming, if you don't already know how to program in this language.

Pascal 3.0 Workstation System — This manual describes in detail all of the subsystems, such as the Editor, Filer, and Compiler. It also describes such topics as how the computer configures itself to access File System peripherals and how to add new peripherals. This manual was formerly the *Pascal 2.0 User's Manual.* The "Getting Started" information (Chapter 1 of the former manual) has been moved to the new *Pascal 3.0 User's Guide.* Two new chapters have been added: Special Configurations and Non-Disc Mass Storage.

Pascal 3.0 Procedure Library — This manual was formerly the *Pascal Procedure Library User's Manual.* It is basically the same as the former manual, except for the removal of the LIF Procedures and Graphics chapters (graphics is now covered in its own separate manual), and the addition of the System Devices and Segmentation Procedures chapters.

Pascal 3.0 Graphics Techniques — This manual is an expanded version of the Graphics chapter of the former *Pascal Procedure Library User's Manual.* It provides several useful techniques that you can use in writing Pascal graphics programs.

HP Pascal Language Reference for Series 200 Computers — This manual describes the HP Standard Pascal language, as well as the implementation dependencies of the Workstation Pascal language.

MC 68000 User's Manual — This manual describes MC68000 processor hardware and instruction set. It is the same manual as shipped with the 2.x Pascal systems. It also covers the 68008 and 68010 processors.

System Devices Procedural Interface

The procedural interface to "system devices" (such as the keyboard, clock, screen, etc.) has been modified. These changes will not affect the way the system looks at the level of *standard* HP Pascal procedures. However, if any of your programs use procedures *below* this uppermost level (such as procedures in an operating system module), then you may have to make some changes. See the System Devices chapter of the *Pascal 3.0 Procedure Library* for complete details.

Additional Computers and Hardware Features Supported by 3.0

- Model 217 (HP 9817)
- Model 237 (HP 9837)

Both Model 217 and Model 237 have a new type of keyboard which requires Pascal 3.0. The keyboard model number is the HP 46020 (the 46021 also works), which uses the HP Human-Interface Link (HP-HIL) to communicate with the computer.

Pascal 3.0 also supports an optional "mouse" input device, which can be connected to the computer through the HP Human Interface Link (HP-HIL). The driver supports using the mouse for cursor-movement input in both horizontal and vertical directions; it also defines the buttons on the mouse as [Return] or [ENTER] and [Select] ([EXECUTE]) keys. You also can access the mouse from your own applications programs; see the System Devices chapter of the *Pascal Procedure Library* manual for details.

Both Models 217 and 237 may also have processor boards with Memory-Management Unit (MMU) hardware; if so, the product numbers have 'U' suffixes (such as HP 9817U and 9837U). If the cache-memory feature is also present, then the MMU hardware increases the execution speed of programs (because the cache-memory feature is automatically enabled by Pascal 3.0).

The Model 237 implements a new type of display hardware: a bit-mapped combined alpha/graphics display with a raster size of 1024 by 768 pixels on a 19-inch diagonal CRT screen.

Additional Peripherals Supported by 3.0

- The new Command Set/'80 (CS80) discs, including the HP 7914, 7933, and 7935 Disc Drives
- New stand-alone DC600 (CS80) Tape Drives; right now this category only includes the HP 9144 Tape Drive
- New Sub-Set/'80 (SS80) floppy discs; right now this category only includes the HP 9122 3.5-inch Double-Sided Floppy discs
- Several new versions of the 913x Hard Discs (V and XV suffix drives)

Additional Cards Supported by 3.0

Here are the new cards that are supported by Pascal 3.0

- HP 98255 EPROM and HP 98253 EPROM Programmer cards, which can be used as mass storage devices (see the Non-Disc Mass Storage chapter of this manual for details)
- HP 98259 Magnetic Bubble Memory cards, which can also be used as mass storage devices (see the Non-Disc Mass Storage chapter of this manual for details)
- HP 98635 Floating-Point Math card (see the description of the FLOAT_HDW Compiler option in the Compiler chapter for details)
- HP 98257 1-Megabyte Memory card, which features parity-checking hardware

Peripheral Configuration

How the system boots and auto-configures itself is fully discussed in the Special Configurations chapter. The Pascal 3.0 TABLE program has even more capabilities than the 2.x version: it automatically configures up to 3 floppy disc drives (dual or single) and at least the first hard disc in the system (up to 10 are potentially possible).

A source-code version of the 3.0 TABLE program (CONFIG:CTABLE.TEXT) is also provided with the 3.0 system. You can look at the program and read the corresponding commentary in the Special Configurations chapter to see exactly how the auto-configuration process works, and you can modify certain portions of it to make your own special configurations. A major change with the 3.0 TABLE is that now you can "coalesce" logical volumes on hard discs without the need to modify and recompile the TABLE source program.

The TABLE program can now easily support printers with RS-232C interfaces by making one small change in the program and recompiling. See the Special Configurations chapter of this manual for details.

Object Code Compatibility

Several internal changes were made with Pascal 3.0. These changes resulted in corresponding changes in the internal representation of object code files. In general, when a version of the system is not compatible with other versions, the leading digit of the version number will be changed. For instance, versions 3.0 and 2.0 are not object-code compatible, while versions 2.1 and 2.0 are.

While it is regrettable, there really is no alternative to these compatibility restrictions. On the positive side, Pascal application programs which don't "fiddle around" in the operating system are source-code compatible with 3.0, so recompilation is usually all that's necessary.

General System Features Added by 3.0

Stream Files: Stream files on read-only devices are now allowed; adding the `[*]` specifier to the stream file name allows this usage by disabling the prompt feature. This same mechanism also allows the use of a stream file called AUTOKEYS to provide "autostart" capabilities with read-only system volumes. See the description of the Stream command in the Overview chapter of this manual for details.

Filer: The Filer can now perform a Translate operation to the CONSOLE: volume, with the ability to view the translated file one screen at a time. See the Filer chapter of this manual for details.

Compiler: The following Compiler options were added. The WARN option allows you to disable warning messages. The FLT_HDW option allows you to specify one of three actions: ON specifies that the Compiler is to emit code that assumes a 98635 Floating-Point Math card is installed in the computer; TEST specifies that the emitted code is to test for the presence of the card; OFF specifies that emitted code always uses floating-point library routines. See the Compiler chapter of this manual for details.

Assembler: The Assembler has been modified to allow use of the new op codes provided by the 68010 processor (such as the MOVES and RTD instructions).

Librarian: A special "edit" mode was added to the Librarian. It allows you to add modules to an existing library more easily. The Librarian can also unassemble the new instructions for the 680xx processors (such as the MOVES and RTD instructions). See the Librarian chapter of this manual for details.

Debugger: These are the new commands that have been added to the Debugger: "X" format for reverse assembly; "R" format for displaying REAL numbers; "O" format and FO default format for octal numbers; added repeat counts on format specifiers; "!" input format for binary numbers; "%" input format for octal numbers; relational operators can now be used in expressions; DA and DG commands for DUMP ALPHA and DUMP GRAPHICS functions, and also the ability to use the corresponding keys; four more softkeys now available (10 total); five more breakpoints now available (9 total); PN and PX commands (PX is an alternate syntax for the existing P command); IF, ELSE, and END commands for conditional execution of Debugger commands added; CALL command added; EC and ETC commands added; [PAUSE] key definition changed. See the Debugger chapter of this manual for details.

Segmentation Procedures: Several procedures that add the capability of run-time program segmentation have been added to the system. See the Segmentation Procedures chapter of the *Pascal 3.0 Procedure Library* manual for details.

Pascal 3.01

The purpose of this revision is to fix bugs in version 3.0 of the Pascal system. The 3.01 BOOT: and ASM: discs contain software which corrects the bugs. (Note that other discs have not been revised).

Note

These revisions do *not* add any features to the system; they only fix bugs in existing features.

Documentation Changes

Since the 3.01 software does not add any features to the system, you may replace references to the 3.0 BOOT: and ASM: discs with references to the 3.01 discs.

Disposition of 3.0 BOOT: and ASM: Discs

If you have version 3.0 BOOT: and ASM: discs, replace them with the 3.01 discs. **Do not use the old discs any longer.**

List of Bugs Fixed

Here are the areas in which bugs have been fixed by the 3.01 revision:

- Flexible disc initialization on Model 226 and 236 Computers equipped with an HP-UX Memory Management processor board and the 3.0 Boot ROM.
- Softkeys and bus errors while using the Debugger.
- Disassembly of shift and rotate instructions with the REVASM module.
- Model 237 display driver module (CRTB)
- Non-advancing characters on some foreign language keyboards.

Pascal 3.1

The main purpose of this version of the system is to add support of Series 300 computers. It also adds support of a few new peripherals, as well as fixes miscellaneous system bugs discovered since the release of Pascal 3.01.

New Computer Hardware Supported by 3.1

New features of the Series 300 computers include the following:

- Many choices of processor, display, and human interface boards:
  - Five new displays (including a separate, high-speed display controller)
  - Two new processors: MC68010, and MC68020 (with MC68881 math co-processor)
  - Battery-backed, real-time clock
  - RS-232C serial interface (similar to the 98644 serial interface)
  - 46020 (and 46021) HP-HIL keyboard (similar to keyboards used with Models 217 and 237, but different from other Series 200 models)
- Support of two new foreign keyboards (Swiss-German and Swiss French).

For a more complete description of the Series 300 enhancements to Series 200 hardware, see the "Porting to Series 300" chapter of the *Pascal Workstation System* manual.

New Peripherals Supported by 3.1

The 3.1 DGL (Device-independent Graphics Library) provides support for the following new HP-HIL (Hewlett-Packard Human Interface Link) devices:

- HP 46087 and 46088 Graphics Tablets ("absolute" graphics input devices)
- HP 35723 TouchScreen (also an "absolute" input device), which attaches to HP 35731 and 35741 Medium-Resolution, 12-inch monitors

Object Code Compatibility

Pascal 3.1 is generally object-code compatible with Pascal 3.0 and 3.01 programs; i.e., programs compiled on the 3.0 or 3.01 systems will generally run on the 3.1 system (with no recompilation required). However, you should not use 3.0 or 3.01 libraries on the 3.1 system. See the "Porting to Series 300" chapter of this manual for further information on how to determine object-code compatibility.

Backward compatibility (i.e., running 3.1-compiled programs on a 3.0 or 3.01 system) is not generally supported. This incompatibility is the result of new run-time support modules that were added for the increases in sizes of SET variables (see the description of new Compiler features below for details regarding the increase in SET size) as well as changes in the interface to the SYSDEVS operating system module.

General System Features Added by 3.1

In general, the new system features provided by Pascal 3.1 are related to the support of the new Series 300 computer hardware, or to new HP-HIL peripherals. Here is a brief list of the features, organized according to subsystem.

Compiler: The new COMPILE20 compiler generates MC68020 instructions, and with the FLOAT_HDW Compiler option supports the use of the MC68881 floating-point co-processor. See the "Compiler" chapter of this manual for details.

A larger SET variable size limit is supported (was 256 elements; is now 262 000 elements). The COMPILER was also modified to fully conform to the HP Pascal Standard (the new COMPILE20 compiler also fully conforms):

- Conformant arrays are now supported.
- Passing elements of packed arrays or records as VAR parameters to procedures or functions is now disallowed (preceding Compiler versions allowed it, although the *HP Pascal Language Reference* showed it as disallowed). You must now use the $ALLOW_PACKED ON$ compiler option if you want to pass this type of parameter.
- You cannot assign values to the index of a FOR loop within the loop (previous versions allowed it).

See the *HP Pascal Language Reference* for details.

Assembler: The Assembler was upgraded to assemble MC68020 and MC68881 instructions. It also supports a new operand syntax which is required to assemble these instructions. See "Instruction Syntax" in the "Assembler" chapter of this manual.

Librarian: The Librarian was upgraded to dis-assemble all MC68020 and MC68881 instructions.

Debugger: The REVASM module was also upgraded to dis-assemble all MC68020 and MC68881 instructions. Since the MC68020 processor has a 32-bit address bus, all addresses specified in the Debugger command line must contain all 32 bits if located in RAM space (see the subsequent "Physical Memory Map" section for details on RAM space bounds).

Error numbers: ESCAPECODE values 30 (arcsin or arccos argument is greater than 1) and 31 (illegal real number) have been added to report MC68881 floating-point math co-processor errors.

IO Library: The IO library has added two registers for the built-in 98644 RS-232C serial interface in Series 300 computers. They allow you to simulate the configuration switches of the built-in 98626 serial interface of the Series 200 Models 216 and 217 computers. See the "RS-232 Serial Interface" chapter of the *Pascal Procedure Library* manual for details.

SYSDEVS interface: This operating system interface module has been modified in the way that highlight characters (130, 131, and 134 thru 143) are displayed in "debugger windows." The variable `debughighlight` indicates which highlight(s) should be applied to characters put in a debugger window using the `dbput` operation. The `dbhighl` operation is a no-op for Series 300 and HP 98700 displays. See the "System Devices" chapter of the *Pascal Procedure Library* manual for details.

Graphics Library: Another version of the Device-independent Graphics Library (DGL) is provided with the system (FLT20:FGRAPH20). It utilizes the MC68020 processor and MC68881 co-processor. (The FGRAPHICS library utilizes the HP 98635 Floating-Point Math Card; the GRAPHICS library uses math libraries.)

System Discs

Three new discs were added to the 3.1 set (single-sided media options), making a total of 14 discs:

- The BOOT2: disc contains the drivers for the Series 300 displays and for the HP 98700 Display Controller. (The BOOT: disc is provided for Series 200 displays and for the HP 98546 Compatibility Video Card Set; see the "Porting to Series 300" chapter for details of using the Compatibility Card.)
- The CMP20: disc contains a compiler that generates object code for the Series 300 computers that feature an MC68020 processor.
- The FLT20: disc contains a new version of the Device-independent Graphics Library (DGL); the file is named FGRAPH20. The set of procedures it provides is the same as the GRAPHICS and FGRAPHICS libraries, but this library contains code that utilizes the MC68020 processor and MC68881 co-processor (instead of the HP 98635 Floating-Point Math Card).

The contents of the following 3.1 discs have changed slightly from their 3.0 counterparts:

- The LIB: disc has only the IO library.
- The CONFIG: disc has a new file containing the new DGL_ABS module (support for the new HP-HIL graphics tablets and TouchScreen, which are "absolute" input devices).

The rest of the 3.1 discs contain the same files as the 3.0 and 3.01 systems:

BOOT:	SYSVOL:	ACCESS:	CMP:	ASM:
GRAPH:	FLTLIB:	DOC:	DGLPRG:	

Furthermore, the Workstation Pascal System is now available on *double-sided, double-density,* 3½-inch, flexible micro-disc media. With this media option, only eight discs are shipped:

Double-Sided Disc	Corresponding Single-Sided Disc(s)
BOOT:	Same files as single-sided BOOT: disc.
BOOT2:	Same files as single-sided BOOT2: disc.
SYSVOL:	Contains files on single-sided SYSVOL:, LIB:, and GRAPH: discs.
ACCESS:	Contains files on single-sided ACCESS: and CONFIG: discs.
CMP:	Contains files on single-sided CMP: and CMP20: discs.
ASM:	Same files as single-sided ASM: disc.
FLTLIB:	Contains files on single-sided FLTLIB: and FLT20: discs.
DOC:	Contains files on single-sided DOC: and DGLPRG: discs.

Documentation

Manuals for the MC68020 processor and MC68881 co-processor have been added to the documentation set.

Pascal User's Guide: This manual has been updated to parallel the structure of the new *Series 200/300 Peripheral Installation Guide*, as well as to discuss adding new peripherals supported by the 3.1 system.

Workstation Pascal System: The "Compiler" chapter describes the new ALLOW_PACKED option, as well as the addition to the FLOAT_HDW option. The "Assembler" chapter has been revised to describe the new addressing modes available with the MC68020 processor. Chapters 10 through 17 were added to describe Pascal programming topics specific to the Workstation System. Chapter 20 describes the considerations you must take in porting existing Pascal programs for Series 200 computers to run on Series 300 computers.

Pascal Procedure Library: The "RS-232C Serial Interface" chapter describes the new registers for the built-in 98644 serial interface in Series 300 computers. The "System Devices" chapter describes the changes to the SYSDEVS interface.

Pascal Graphics Techniques: The "Interactive Graphics" chapter describes the new HIL input devices (graphics tablets and TouchScreen). The "Color Graphics" chapter describes the use of the new color displays. The "Procedure Reference" section has been updated accordingly.

HP Pascal Language Reference: The "Compiler Options" section of the "Workstation Implementation" appendix describes the new ALLOW_PACKED option, as well as the addition to the FLOAT_HDW option.

All Pascal manuals have new part numbers with this revision of the system.

Pascal 3.12

The sole purpose of this revision of the Pascal operating system was to add support of the HP 98203C keyboard to the list of supported devices.

Hardware Differences

The two hardware differences between this keyboard and the HP 98203B keyboard (and the built-in keyboards of the Model 226 and 236 computers) are:

- This keyboard is connected to the computer through the HP Human Interface Link (HP-HIL), rather than through the HP 98203B-type keyboard interface.
- This keyboard's built-in knob operates like the *separate* HP-HIL knob, rather than the built-in knob on the HP 98203B keyboards.

Software Differences

In order to use the HP 98203C keyboard and knob, you must have the following Pascal operating system components:

- The 3.12 (or later) versions of the SYSDEVS and A804XDVR operating system modules.
- The 3.12 (or later) versions of the MOUSE and HPHIL driver modules.
- The 3.12 (or later) version of the STARTUP file (optional).

Note that the HPHIL and Mouse driver modules are required *only* if you will be using the knob; they are not needed for general use of the HP 98203C keyboard.

Pascal 3.2

The main purpose of this revision is to add support of a hierarchical file system (HFS) to the system. Other changes include the addition of utilities supporting various aspects of HFS, the support of new peripherals, and various bug fixes of errors discovered since the release of 3.1.

New Computer Hardware Supported by 3.2

- HP98203C HP-HIL keyboard (similar to the HP98203B with RPG knob, but knob is HIL controlled.

New Peripherals Supported by 3.2

- 2227A Thermal ink-jet printer
- 2228A Thermal ink-jet printer
- 3630A Printer/Plotter
- 7570A 8-pen plotter
- 7595A 8-pen plotter
- 7596A 8-pen plotter (roll feed version of 7595A)
- 9153B 20M byte disc
- 7957/7958 discs
- 7936/7937 discs
- 45911A HP-HIL Tablet
- 7907A Fixed/Removable disc

Object Code Compatibility

Pascal 3.2 is generally object-code compatible with Pascal 3.1(3.12) programs; i.e. programs compiled on the 3.1 systems will generally run on the 3.2 system (with no recompilation required).

General System Features Added by 3.2

Most new system features provided by Pascal 3.2 are related to the support of HFS and the ability to transfer files easily between BASIC, HP-UX and Pascal environments. Here is a brief list of the features, organized according to subsystem.

Compiler: While no new features were added to the compiler in this release, the compiler was modified to ensure greater adherence to the HP Pascal Standard.

Assembler: No new features were added in this release.

Librarian: No new features were added in this release.

Debugger: No new features were added in this release.

Error numbers: Five new I/O System Errors have been added to cover the possible error conditions within the HFS DAM. Some existing I/O System Error result values have been modified to suit HFS.

Filer: A new prompt covering HFS access rights for files and directories with its own sub-menu is added to the Filer prompt.

Graphics Library: A new high-performance driver for the HP-HIL Mouse is supplied with 3.2 which also covers the operation of the wheel (or knob) on the now supported 98203C keyboard. This keyboard is an HIL version of the 98203B keyboard, or large keyboard normally supplied with the Model 236.

System Discs

In addition to the regular system discs, HFS support is provided on the HFS: disc (or, if the system was purchased with the single-sided media option, support is provided on the HFS1:, HFS2:, and HFS3: discs).

32-bit Computers

Pascal 3.2 supports both the Model 330 and 350. These computers have true 32-bit addressing capability.

Pascal 3.21

New Features

This revision of Pascal adds support of the following hardware:

- Three new displays: HP98548, HP98549, HP98550 Display Interfaces.
- The Model 319C+ Workstation computer.

Change to Existing Feature

The Pascal system uses a solid-block alpha-cursor rather than an underline with the 98548, 98549, and 98550 displays (the 98549 is the standard display used with 319C+ computers). This cursor implementation makes it easier for you to find the cursor on a screen of text.

Also note that the color of the cursor tracks the "current text color". (The "current text color" is the color that a subsequently displayed character will have; it is **not** necessarily the color of the character under the cursor. For instance, if the current text color is red and you move the cursor over a green character, the cursor will remain red.)

Manuals Updated

This table lists which manuals have been updated to document the 3.21 revision of Pascal.

Manual Title	Changes/Additions	Location in Manual
Pascal Workstation System, Volume 2 (this manual)	Add info about new display drivers (CRTE); Add info describing Pascal 3.21.	Chapter 18; This section.
Pascal Procedure Library	Add info about new displays: DISPLAY_INIT, OUTPUT_ESC, SET_COLOR_TABLE, SET_DISPLAY_LIM, SET_LOCATOR_LIM	See entries in the "Reference" section
Pascal Graphics Techniques	Add info about new displays (same procedures as listed above in the *Procedure Library* manual)	See entries in the "Reference" section

Object-Code Compatibility

Pascal 3.21 is generally object-code compatible with Pascal 3.2; that is, programs compiled on the 3.2 revision of the system will generally run on the 3.21 revision (with no recompilation required).

System Discs Modified

The following **single**-sided discs have been modified by the 3.21 revision:

BOOT2: The CRTE module has been added to INITLIB; it contains drivers for the new displays.

GRAPH: The GRAPHICS file has been recompiled to support the HP98548A, HP98549A, and HP98550A displays.

FLTLIB: The FGRAPHICS file has been recompiled to support the HP98548A, HP98549A, and HP98550A displays.

FLT20: The FGRAPH20 file has been recompiled to support the HP98548A, HP98549A, and HP98550A displays.

The following **double**-sided discs have been modified by the 3.21 revision:

BOOT2:	The CRTE module has been added to INITLIB; it contains drivers for these new displays.
SYSVOL:	The GRAPHICS file has been recompiled to support the HP98548A, HP98549A, and HP98550A displays.
FLTLIB:	The FGRAPHICS and FGRAPH20 files have been recompiled to support the HP98548A, HP98549A, and HP98550A displays.

Pascal 3.22

This revision of Pascal adds support for Models 332, 340, 360 and 370. Also added is support for the VMEbus Interface which is usable with all revisions back to 3.1. Pascal 3.22 also includes support for programmable system reboot, callable from both programs and the Debugger. Programs compiled with the 3.2 and 3.21 versions will generally run on 3.22 with no recompilation required.

New Hardware

- MC68030 and MC68882 processors
- SPU models 332, 340, 360, and 370

New Peripherals

- Plotter models 7575A and 7576A (DraftPro DXL and EXL)

Software Changes

- `System date` is now valid up to the year 2027 but invalid before 1 January 1970.
- The HFS file system now supports changing the parameters in the `MKHFS` program.
- The Debugger's `sb` command has been extended to provide "named" reboot.
- The SYSBOOT library has been added to support programmatic reboot.
- The LAN driver has been added to support the 98643A and built-in LAN interfaces.
- VMELIBRARY has been added to support the 98646A VME Interface card.

System Discs

The following changes were made:

Single-sided discs	LIB: new files are LAN and VMELIBRARY CONFIG: new file is SYSBOOT
Double-sided discs	ACCESS: new files are LAN, VMELIBRARY, and SYSBOOT

Pascal 3.23

This release adds support for the Model 345 and 375 SPUs, including the built in SCSI and PARALLEL interfaces provided with these SPUs. The built in SCSI disc provided with the Model 345 and the upgraded Model 340 are also supported.

SCSI support has been extended to add HP SCSI disc offerings, and also to include a programmer's interface.

Support for the HP parallel interface includes printer support and programmer's support through the existing IO Library. HP ScanJet™ bidirectional-parallel protocol is also supported.

New Hardware

- SPU Models 345 and 375
- HP 98658A DIO I SCSI host adapter
- HP 98265A SCSI single-ended board

New Peripherals

- Models 340 and 345 built-in SCSI discs
- HP SCSI fixed-disc Models 7957S, 7958S, 7959S, C2212A, C2213A
- HP SCSI optical disc Model C1701A (6300 650/A)
- Plotter Models 7595B SX, 7595B RX, and 7599A MX in HPGL compatibility mode only.

Object Code Compatibility

Pascal 3.23 is generally upward code compatible with 3.2 systems. Programs compiled with the 3.2, 3.21, or 3.22 systems should run on 3.23 with no recompilation required. Note that this may not apply if the application contains linked in system modules.

Software Changes

- The CTABLE program and other utilities now recognize SCSI discs with HFS and LIF directory-access methods.
- A SCSI bus driver (SCSIDVR) has been added to provide support of the HP 98658A and HP 98265A interfaces.
- A SCSI disc driver (SCSIDISC) has been added to provide support for the SCSI discs mentioned above.
- A SCSI programmer's interface (SCSILIB) has been added.
- An HP parallel-interface driver (PARALLEL) is added to provide programmer support through the standard I/O library. A new module has been added, PARALLEL_3, to provide extensions to the I/O library.
- The CTABLE program has been updated so that with minor programmer modifications it can be made to recognize parallel printers.
- The PRINTER module is modified for parallel-printer support.

System Discs

The following changes were made:

Single-sided discs:	BUBBLE, EPROMS, EDRIVER, and ETU.CODE were moved from the CONFIG: disc to the LIB: disc.
	SCSIDVR, SCSIDISC, SCSILIB, and PARALLEL were added to the LIB: disc.
	SCSITEST.TEXT and PSCAN.TEXT were added to the DOC: disc.
Double-sided discs:	VMELIBRARY was moved from the ACCESS: disc to the SYSVOL: disc.
	SCSIDVR, SCSIDISC, and PARALLEL were added to the ACCESS: disc.
	SCSILIB was added to the SYSVOL: disc.
	SCSITEST.TEXT and PSCAN.TEXT were added to the DOC: disc.

File Interchange Between Pascal and BASIC

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

- Pascal and BASIC treat LIF directories on flexible discs similarly. ASCII text files are intended to be used as the interchange mechanism.
- It was mentioned earlier that Pascal compresses the suffix of user file names on LIF discs in order to effectively allow longer file names. BASIC doesn't know about compressed names, so the BASIC program needs to invert the compression algorithm. This inversion is very simple, and is described in the section of the File System chapter called Programming with Files. Essentially, Pascal chops off the dot and the suffix (such as .ASC), then appends the first letter of the suffix and enough trailing "_" characters to make a 10-character name. Thus "ABC.ASC" becomes "ABCA_____", which is the name BASIC will see.
- BASIC can't deal with more than one LIF directory on a hard disc. When using LIF, Pascal wants to divide large hard discs into several volumes, each with its own directory (you can override this by modifying CTABLE). Hard disc partitioning is described in the Special Configurations chapter.

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

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

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

Module Names Used by the Operating System

The file named SYMBOLS.TEXT on the DOC: disc contains the names of modules and symbols that are present in the system as it is shipped from the factory. They are provided so that you will not name a module using any of these names, unless you *definitely* want to override the system module's function.

Note that many of these module names do not show up in the system symbol table (for example, they may have been removed by linking). However, you should not use them, because HP reserves the right to use them in the future.

Physical Memory Map

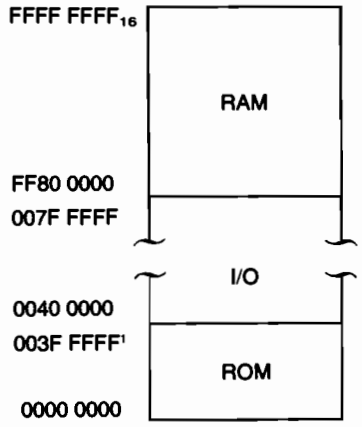
The first part of this section describes the physical hardware memory map of your Series 200/300 computer with regard to ROM space, I/O space and RAM space. This section begins with an overview of the hardware memory layout, followed by a more detailed memory map of each major section of memory.

Register addresses and descriptions are included for the internal I/O devices.

16 Megabyte Address Range

For Series 200 and Models 310 and 320

There are 23 address lines (BA1..BA23) providing 16 megabyte addressing on WORD boundaries. For byte operations, two control lines BUDS (Buffered Upper Data Strobe) and BLDS (Buffered Lower Data Strobe) indicate whether the upper data byte (BD8 through BD15), the lower data byte (BD0) through BD7), or both bytes are involved in the communication. Note: When BA0) = 0, the high byte is requested. When BA0 = 1, the lower byte is requested.



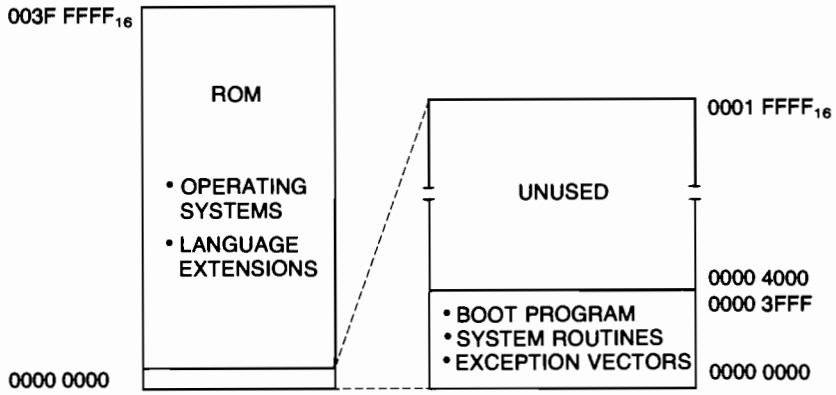
For Models 330, 332, 340, 345, 350, 360, 370, and 375

The Pascal Workstation does not use the virtual memory capability of these models. Instead, all available RAM is used as a single, linear address space.

Note

For a complete description of the HP Series 200 and 300 computer's physical memory maps, see the *Pascal System Internals* documentation for revision 3.1.

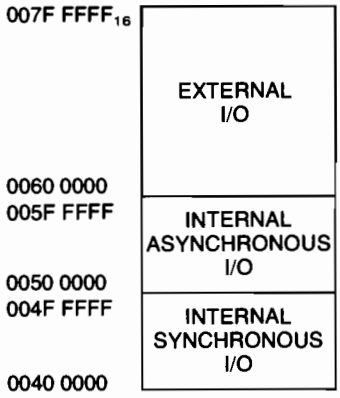
The Overall ROM Memory Map



The boot program, exception vectors, and some system routines reside on ROM chips starting at $000000 and extending to $003FFF. The space between $004000 and $01FFFF is unused. (The BOOTROM 3.0 consists of approximately 48K bytes of code, starting at $000000). The boot program checks for system ROMs and language extension ROMs on 16K boundaries beginning at $020000 and continuing up to $3FC000. These ROMs are recognized by their appropriate header information.

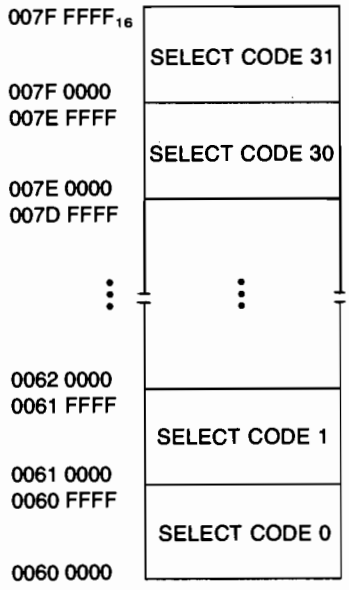
Memory Mapped I/O

I/O memory space is divided into three sections. External I/O is that section which corresponds to the backplane of the Series 200/300 computers. The select codes on the backplane I/O cards correspond to address bits BA16 through BA20.

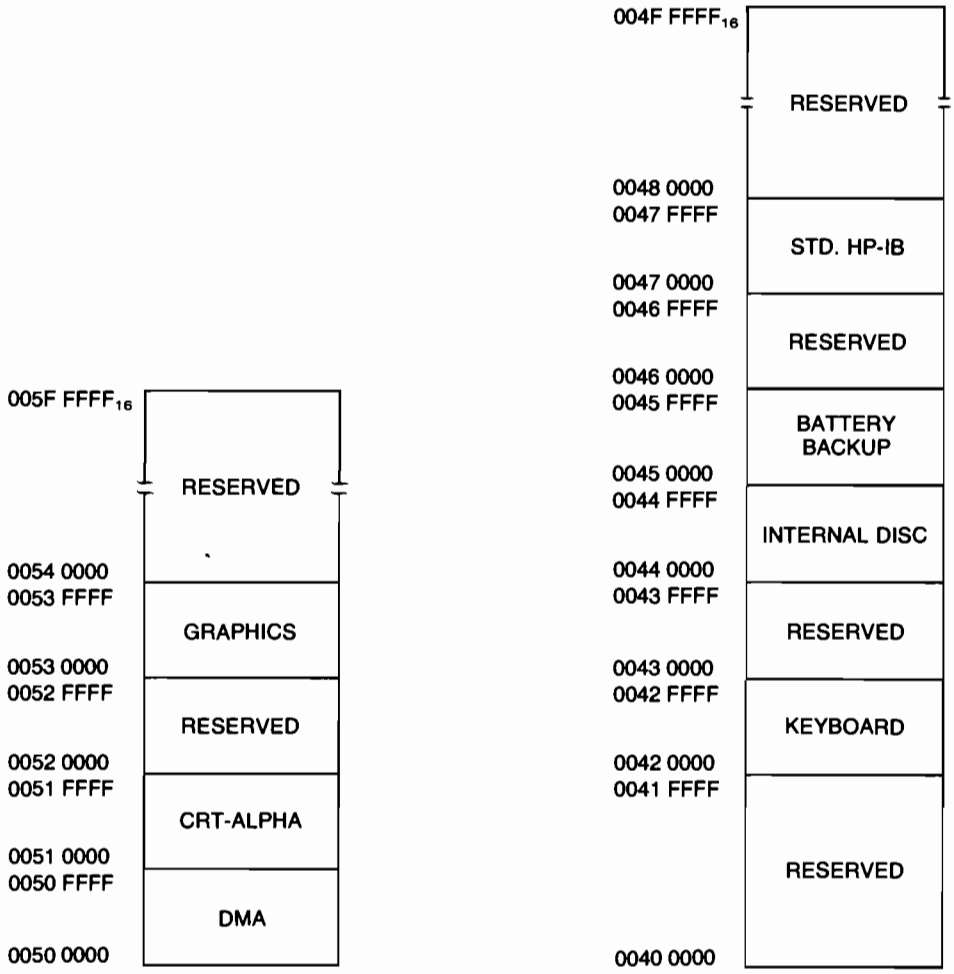


External I/O

All supported I/O cards available for the Series 200/300 computers are mapped into the external I/O space on 64K boundaries (except the DMA card which maps with synchronous internal I/O). There are 32 such spaces between $600000 and $7FFFFF. User designed cards must also map into one of these spaces. The select codes correspond to address lines BA16 through BA20. HP cards have been assigned default select codes but can be reset by the user to map into any configuration.



Internal I/O



Internal I/O functions are doubly mapped, once between $400000 and $4FFFFF, and again between $500000 and $5FFFFF. That is why much of the internal I/O space is reserved. Those I/O functions can be addressed in either range. The difference is that the low range generates a DTACK (Data Acknowledge) automatically, whether the card has actually responded to the data or not. If the synchronous cards are addressed in the high range, there will be no DTACK generated at all.

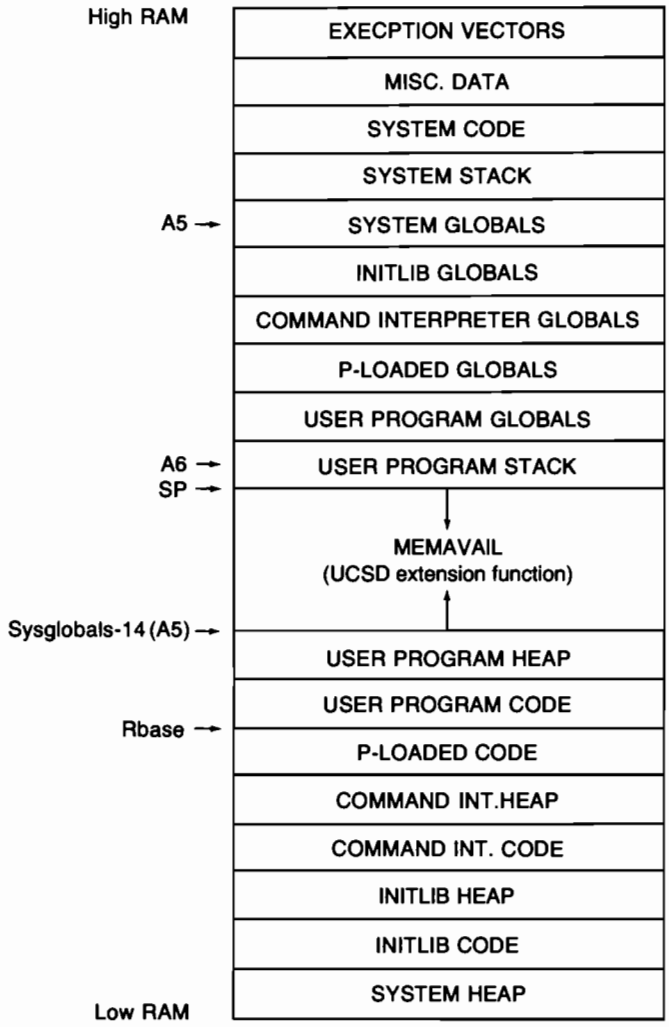
The Software Memory Map

This software memory map shows the symbolic locations of global variables, local variables, the stack, the heap and the code relocation base.

If you'll look at the directory of the boot disk, you'll see the files that are loaded into RAM at power-up. SYSTEM_P is loaded first. It shows as SYSTEM CODE on the map[1]. It sets up the EXCEPTION VECTORS, MISCelaneous DATA, SYSTEM GLOBALS, SYSTEM STACK, INITLIB GLOBALS and SYSTEM HEAP. TABLE is loaded and it modifies some of the data in SYSTEM GLOBALS and SYSTEM HEAP.

INITLIB is then loaded. It contains the I/O drivers, file system drivers, etc.

Finally, STARTUP is loaded. When you receive a Pascal Language System from HP, STARTUP is the Main Command Level command interpreter. It handles P-loading files, loading subsystem files, and loading user programs. You may use the Filer to change any file you want to STARTUP and that file will execute at power-up. A copy of the BOOT disc should be created before the change is made because you won't be able to change the file back to the original configuration.



Note

For a complete description of the HP Series 200 Workstation Pascal's software memory map, see the *Pascal System Internals* documentation.

[1] When booting from HFS discs, SYSTEM_P is initially loaded near the bottom of RAM. SYSTEM_P checks for this condition, and moves itself if necessary, before beginning its "real" execution.

Character Sets

C

This section provides tables for the following character sets:

- U.S. ASCII character set
- U.S./European display characters (for Models 216, 220, 226, and 236 Computers)
- U.S./European display characters (for Models 217 and 237 and Series 300 Computers)
- Katakana display characters (for all Series 200 Computers)
- CRT highlight characters (for the Model 236 Computer)

U.S. ASCII Character Set

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(	40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
-	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

STD-LL-60182

U.S. ASCII Character Set

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[	91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
|	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

U.S./European Display Characters

Display characters for the Model 216, 220, 226, and 236 Computers.

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
^@	0	00000000
^A	1	00000001
^B	2	00000010
^C	3	00000011
^D	4	00000100
^E	5	00000101
^F	6	00000110
^G	7	00000111
^H	8	00001000
^I	9	00001001
^J	10	00001010
^K	11	00001011
^L	12	00001100
^M	13	00001101
^N	14	00001110
^O	15	00001111
^P	16	00010000
^Q	17	00010001
^R	18	00010010
^S	19	00010011
^T	20	00010100
^U	21	00010101
^V	22	00010110
^W	23	00010111
^X	24	00011000
^Y	25	00011001
^Z	26	00011010
^[	27	00011011
^\	28	00011100
^]	29	00011101
^^	30	00011110
^_	31	00011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
	32	00100000
!	33	00100001
"	34	00100010
#	35	00100011
$	36	00100100
%	37	00100101
&	38	00100110
'	39	00100111
(	40	00101000
)	41	00101001
*	42	00101010
+	43	00101011
,	44	00101100
-	45	00101101
.	46	00101110
/	47	00101111
0	48	00110000
1	49	00110001
2	50	00110010
3	51	00110011
4	52	00110100
5	53	00110101
6	54	00110110
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010
[	91	01011011
\	92	01011100
]	93	01011101
^	94	01011110
_	95	01011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
`	96	01100000
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010
{	123	01111011
|	124	01111100
}	125	01111101
~	126	01111110
▒	127	01111111

STD-L-90182

U.S./European Display Characters

Display characters for the Model 216, 220, 226, and 236 Computers.

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
NOTE	128	10000000
NOTE	129	10000001
NOTE	130	10000010
NOTE	131	10000011
NOTE	132	10000100
NOTE	133	10000101
NOTE	134	10000110
NOTE	135	10000111
NOTE	136	10001000
NOTE	137	10001001
NOTE	138	10001010
NOTE	139	10001011
NOTE	140	10001100
NOTE	141	10001101
NOTE	142	10001110
NOTE	143	10001111
▯	144	10010000
▯	145	10010001
▯	146	10010010
▯	147	10010011
▯	148	10010100
▯	149	10010101
▯	150	10010110
▯	151	10010111
▯	152	10011000
▯	153	10011001
▯	154	10011010
▯	155	10011011
▯	156	10011100
▯	157	10011101
▯	158	10011110
▯	159	10011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
▯	160	10100000
▯	161	10100001
▯	162	10100010
▯	163	10100011
▯	164	10100100
▯	165	10100101
▯	166	10100110
▯	167	10100111
¨	168	10101000
`	169	10101001
^	170	10101010
¨	171	10101011
~	172	10101100
▯	173	10101101
▯	174	10101110
£	175	10101111
—	176	10110000
▯	177	10110001
▯	178	10110010
°	179	10110011
▯	180	10110100
ç	181	10110101
Ñ	182	10110110
ñ	183	10110111
¡	184	10111000
¿	185	10111001
¤	186	10111010
£	187	10111011
▯	188	10111100
§	189	10111101
▯	190	10111110
▯	191	10111111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
à	192	11000000
ê	193	11000001
ô	194	11000010
û	195	11000011
á	196	11000100
é	197	11000101
ó	198	11000110
ú	199	11000111
à	200	11001000
è	201	11001001
ò	202	11001010
ù	203	11001011
ä	204	11001100
ë	205	11001101
ö	206	11001110
ü	207	11001111
Å	208	11010000
î	209	11010001
Ø	210	11010010
Æ	211	11010011
å	212	11010100
í	213	11010101
ø	214	11010110
æ	215	11010111
Ä	216	11011000
ì	217	11011001
Ö	218	11011010
Ü	219	11011011
É	220	11011100
ï	221	11011101
ß	222	11011110
▯	223	11011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
▯	224	11100000
▯	225	11100001
▯	226	11100010
▯	227	11100011
▯	228	11100100
▯	229	11100101
▯	230	11100110
▯	231	11100111
▯	232	11101000
▯	233	11101001
▯	234	11101010
▯	235	11101011
▯	236	11101100
▯	237	11101101
▯	238	11101110
▯	239	11101111
▯	240	11110000
▯	241	11110001
▯	242	11110010
▯	243	11110011
▯	244	11110100
▯	245	11110101
▯	246	11110110
▯	247	11110111
▯	248	11111000
▯	249	11111001
▯	250	11111010
▯	251	11111011
▯	252	11111100
▯	253	11111101
▯	254	11111110
▯	255	11111111

STD-LL-40182

NOTE: Ignored by the 9826; see "Monochrome Highlight Characters."

U.S./European Display Characters

Display characters for the Model 217 and 237 Computers.

ASCII

Num.	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
0	^U	32		64	@	96	`
1	^H	33	!	65	A	97	a
2	^X	34	"	66	B	98	b
3	^X	35	#	67	C	99	c
4	^T	36	$	68	D	100	d
5	^O	37	%	69	E	101	e
6	^K	38	&	70	F	102	f
7	^G	39	'	71	G	103	g
8	^S	40	(	72	H	104	h
9	^T	41	)	73	I	105	i
10	^F	42	*	74	J	106	j
11	^T	43	+	75	K	107	k
12	^F	44	,	76	L	108	l
13	^R	45	-	77	M	109	m
14	^O	46	.	78	N	110	n
15	^I	47	/	79	O	111	o
16	^L	48	0	80	P	112	p
17	^1	49	1	81	Q	113	q
18	^2	50	2	82	R	114	r
19	^3	51	3	83	S	115	s
20	^4	52	4	84	T	116	t
21	^K	53	5	85	U	117	u
22	^Y	54	6	86	V	118	v
23	^B	55	7	87	W	119	w
24	^N	56	8	88	X	120	x
25	^H	57	9	89	Y	121	y
26	^B	58	:	90	Z	122	z
27	^L	59	;	91	[	123	{
28	^S	60	<	92	\	124	|
29	^S	61	=	93	]	125	}
30	^^	62	>	94	^	126	~
31	^_	63	?	95	_	127	■

U.S./European Display Characters

Display characters for the Model 217 and 237 Computers.

ASCII

Num.	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
128	CL	160		192	â	224	Á
129	IV	161	À	193	ê	225	Â
130	BG	162	Â	194	ô	226	ã
131	IB	163	È	195	û	227	Ð
132	UL	164	ê	196	á	228	đ
133	IK	165	Ë	197	é	229	Đ
134	BG IK	166	î	198	ó	230	Ð
135	IK	167	ï	199	ú	231	ó
136	HH	168	´	200	à	232	Ò
137	RD	169	`	201	è	233	Õ
138	YE	170	^	202	ò	234	õ
139	GR	171	¨	203	ù	235	Š
140	CY	172	~	204	ä	236	š
141	BU	173	Ù	205	ë	237	Ú
142	MG	174	Ô	206	ö	238	Ÿ
143	BK	175	£	207	ü	239	ÿ
144	90	176	¯	208	Å	240	Þ
145	91	177	B1	209	î	241	þ
146	92	178	B2	210	Ø	242	F2
147	93	179	·	211	Æ	243	F3
148	94	180	Ç	212	å	244	F4
149	95	181	ç	213	í	245	I0
150	96	182	Ñ	214	ø	246	–
151	97	183	ñ	215	æ	247	¼
152	98	184	¡	216	Ä	248	½
153	99	185	¿	217	ì	249	ª
154	9A	186	¤	218	Ö	250	º
155	9B	187	£	219	Ü	251	«
156	9C	188	¥	220	É	252	■
157	9D	189	§	221	ï	253	»
158	9E	190	ƒ	222	ß	254	±
159	9F	191	¢	223	Ô	255	▯

U.S./European Display Characters

Display characters for the Series 300 Computers.

ASCII

Num.	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
0	NU	32		64	@	96	`
1	SH	33	!	65	A	97	a
2	SX	34	"	66	B	98	b
3	EX	35	#	67	C	99	c
4	ET	36	$	68	D	100	d
5	EQ	37	%	69	E	101	e
6	AK	38	&	70	F	102	f
7	BL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DL	48	0	80	P	112	p
17	D1	49	1	81	Q	113	q
18	D2	50	2	82	R	114	r
19	D3	51	3	83	S	115	s
20	D4	52	4	84	T	116	t
21	NK	53	5	85	U	117	u
22	SY	54	6	86	V	118	v
23	EB	55	7	87	W	119	w
24	CN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SB	58	:	90	Z	122	z
27	EC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	■

U.S./European Display Characters

Display characters for the Series 300 Computers.

ASCII

Num.	Chr.	Num.	Chr.	Num.	Chr.	Num.	Chr.
128	^@	160		192	À	224	Á
129	^A	161	À	193	Ê	225	Ã
130	^B	162	Â	194	Ô	226	ã
131	^C	163	È	195	Û	227	Đ
132	^D	164	Ê	196	á	228	đ
133	^E	165	Ë	197	é	229	Ĩ
134	^F	166	Î	198	ó	230	ĩ
135	^G	167	Ï	199	ú	231	Ó
136	^H	168	´	200	à	232	Ò
137	^I	169	`	201	è	233	Õ
138	^J	170	^	202	ò	234	õ
139	^K	171	¨	203	ù	235	Š
140	^L	172	~	204	ä	236	š
141	^M	173	Ù	205	ë	237	Ú
142	^N	174	Ô	206	ö	238	Ÿ
143	^O	175	£	207	ü	239	ÿ
144	^P	176	¯	208	Á	240	Þ
145	^Q	177	Ý	209	Î	241	þ
146	^R	178	ý	210	Ø	242	·
147	^S	179	·	211	Æ	243	µ
148	^T	180	Ç	212	å	244	¶
149	^U	181	ç	213	í	245	¼
150	^V	182	Ñ	214	ø	246	–
151	^W	183	ñ	215	æ	247	¼
152	^X	184	¡	216	Ä	248	½
153	^Y	185	¿	217	ì	249	Å
154	^Z	186	¤	218	Ö	250	º
155	^[	187	£	219	Ü	251	«
156	^\	188	¥	220	É	252	■
157	^]	189	§	221	ï	253	»
158	^^	190	ƒ	222	ß	254	±
159	^_	191	¢	223	Ô	255	▮

Katakana Display Characters

Display characters for all Series 200 Computers (while in Katakana mode).

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
▯	0	00000000
▯	1	00000001
▯	2	00000010
▯	3	00000011
▯	4	00000100
▯	5	00000101
▯	6	00000110
▯	7	00000111
▯	8	00001000
▯	9	00001001
▯	10	00001010
▯	11	00001011
▯	12	00001100
▯	13	00001101
▯	14	00001110
▯	15	00001111
▯	16	00010000
▯	17	00010001
▯	18	00010010
▯	19	00010011
▯	20	00010100
▯	21	00010101
▯	22	00010110
▯	23	00010111
▯	24	00011000
▯	25	00011001
▯	26	00011010
▯	27	00011011
▯	28	00011100
▯	29	00011101
▯	30	00011110
▯	31	00011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
	32	00100000
!	33	00100001
"	34	00100010
#	35	00100011
$	36	00100100
%	37	00100101
&	38	00100110
'	39	00100111
(	40	00101000
)	41	00101001
*	42	00101010
+	43	00101011
,	44	00101100
-	45	00101101
.	46	00101110
/	47	00101111
0	48	00110000
1	49	00110001
2	50	00110010
3	51	00110011
4	52	00110100
5	53	00110101
6	54	00110110
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010
[	91	01011011
¥	92	01011100
]	93	01011101
^	94	01011110
_	95	01011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
`	96	01100000
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010
{	123	01111011
|	124	01111100
}	125	01111101
~	126	01111110
▯	127	01111111

STD-LL-60182

Katakana Display Characters

Display characters for all Series 200 Computers (while in Katakana mode).

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
NOTE	128	10000000
NOTE	129	10000001
NOTE	130	10000010
NOTE	131	10000011
NOTE	132	10000100
NOTE	133	10000101
NOTE	134	10000110
NOTE	135	10000111
NOTE	136	10001000
NOTE	137	10001001
NOTE	138	10001010
NOTE	139	10001011
NOTE	140	10001100
NOTE	141	10001101
NOTE	142	10001110
NOTE	143	10001111
NOTE	144	10010000
NOTE	145	10010001
NOTE	146	10010010
NOTE	147	10010011
NOTE	148	10010100
NOTE	149	10010101
NOTE	150	10010110
NOTE	151	10010111
NOTE	152	10011000
NOTE	153	10011001
NOTE	154	10011010
NOTE	155	10011011
NOTE	156	10011100
NOTE	157	10011101
NOTE	158	10011110
NOTE	159	10011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
｠	160	10100000
｡	161	10100001
｢	162	10100010
｣	163	10100011
､	164	10100100
･	165	10100101
ｦ	166	10100110
ｧ	167	10100111
ｨ	168	10101000
ｩ	169	10101001
ｪ	170	10101010
ｫ	171	10101011
ｬ	172	10101100
ｭ	173	10101101
ｮ	174	10101110
ｯ	175	10101111
ｰ	176	10110000
ｱ	177	10110001
ｲ	178	10110010
ｳ	179	10110011
ｴ	180	10110100
ｵ	181	10110101
ｶ	182	10110110
ｷ	183	10110111
ｸ	184	10111000
ｹ	185	10111001
ｺ	186	10111010
ｻ	187	10111011
ｼ	188	10111100
ｽ	189	10111101
ｾ	190	10111110
ｿ	191	10111111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
ﾀ	192	11000000
ﾁ	193	11000001
ﾂ	194	11000010
ﾃ	195	11000011
ﾄ	196	11000100
ﾅ	197	11000101
ﾆ	198	11000110
ﾇ	199	11000111
ﾈ	200	11001000
ﾉ	201	11001001
ﾊ	202	11001010
ﾋ	203	11001011
ﾌ	204	11001100
ﾍ	205	11001101
ﾎ	206	11001110
ﾏ	207	11001111
ﾐ	208	11010000
ﾑ	209	11010001
ﾒ	210	11010010
ﾓ	211	11010011
ﾔ	212	11010100
ﾕ	213	11010101
ﾖ	214	11010110
ﾗ	215	11010111
ﾘ	216	11011000
ﾙ	217	11011001
ﾚ	218	11011010
ﾛ	219	11011011
ﾜ	220	11011100
ﾝ	221	11011101
ﾞ	222	11011110
ﾟ	223	11011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
ﾠ	224	11100000
ﾡ	225	11100001
ﾢ	226	11100010
ﾣ	227	11100011
ﾤ	228	11100100
ﾥ	229	11100101
ﾦ	230	11100110
ﾧ	231	11100111
ﾨ	232	11101000
ﾩ	233	11101001
ﾪ	234	11101010
ﾫ	235	11101011
ﾬ	236	11101100
ﾭ	237	11101101
ﾮ	238	11101110
ﾯ	239	11101111
ﾰ	240	11110000
ﾱ	241	11110001
ﾲ	242	11110010
ﾳ	243	11110011
ﾴ	244	11110100
ﾵ	245	11110101
ﾶ	246	11110110
ﾷ	247	11110111
ﾸ	248	11111000
ﾹ	249	11111001
ﾺ	250	11111010
ﾻ	251	11111011
ﾼ	252	11111100
ﾽ	253	11111101
ﾾ	254	11111110
■	255	11111111

STD-LL-60182

NOTE: These are the same as the U.S./European characters.

Monochrome Highlight Characters

These characters affect the highlight mode on all subsequently displayed characters on monochrome displays (not implemented on some Model 216 and 220 displays). Note that bit-mapped alpha displays have no blinking or half-bright modes (Model 237 and all Series 300 displays).

ASCII code	Inverse Video	Blinking	Underline	Halfbright
128				
129	X			
130		X		
131	X	X		
132			X	
133	X		X	
134		X	X	
135	X	X	X	
136				X
137	X			X
138		X		X
139	X	X		X
140			X	X
141	X		X	X
142		X	X	X
143	X	X	X	X

In the table above, "X" means the highlight is enabled by displaying the ASCII character.

Color Highlight Characters

These characters change the color of subsequently displayed characters on the alpha screen of color displays.

ASCII code	Color	Pen
136	white	1
137	red	2
138	yellow	3
139	green	4
140	cyan	5
141	blue	6
142	magenta	7
143	black	0

Note that the colors shown in this table are the default colors for the corresponding pen number. On Series 300 displays, changing one of these pen colors also affects the color corresponding to the character. However, on the Model 236C, changing one of these pen colors has no effect on the alpha display character's color.

Command Summaries

D

Main Command Level Summary

Assembler	Run the Assembler.
Compiler	Run the Pascal Compiler.
Debugger	Run the Debugger subsystem.
Editor	Run the Editor subsystem.
e**X**ecute	Execute a specified object file.
Filer	Run the Filer subsystem.
Initialize	Places all current blocked devices on-line.
Librarian	Run the Librarian subsystem.
Memvol	Create a memory resident volume.
Newsys	Select a new System Volume.
Permanent	Move an object file from a mass storage medium into internal memory.
Run	Compile and execute the workfile or execute the last file compiled.
Stream	Stream a text file to be processed as keyboard commands.
User restart	Run the last program executed.
Version	Display version information about the Pascal Operating System.
What	Display the complete file specifier of each Pascal subsystem.
?	Display alternate command prompt.

Editor Command Summary

Text Modifying Commands

Copy	Insert text from the copy buffer or from an external file in front of the current cursor location.
Delete	Remove text from the current cursor location to the location of the cursor when Select (EXECUTE) is pressed.
Insert	Inserts text in front of the current cursor location.
Replace	Replace the specified target string with the specified substitute string.
e**X**change	Replace the text at the cursor with text typed from the keyboard, on a character-by-character basis.
Zap	Delete all text between the anchor and the current cursor location. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)

Text Formatting Commands

Adjust	Adjust the column in which a line (or lines) start.
Margin	Format the paragraph where the cursor is located to the margins in the current environment.

Miscellaneous Commands

Quit	Leave the Editor in an orderly manner. Provides various options for saving the text currently in memory.
Stop (Shift-CLR I/O)	Terminate the Editor subsystem (note that text is lost).
Set	Modify the environment or set markers in the text.
Verify	Update the displayed text to reflect the text stored in memory.

Cursor Keys

Tab	Move cursor to next tab position (fixed tabs) in the current direction.
Return or Enter	Move cursor in current direction to the leftmost character in the next line.
Space bar	Move cursor one character in the current direction.
Arrow keys	Move cursor in the direction specified by the key.
Cursor wheel	Moves the cursor like the arrow keys, but provides user controllable scrolling speed. Without the Shift key, works like right and left arrows; with the Shift key, works like the up and down arrows.

Cursor Positioning Commands

=	The = key positions the cursor at the anchor. (The anchor is set at the location of the latest Adjust, Find, Insert, or Replace command.)
Find	Position the cursor after the specified target string.
Jump	Position the cursor at the beginning, end, or at the specified marker.
Page	Position the cursor ± 1 page from the current location.

Filer Command Summary

Volume Related Commands

Bad sectors	Scans a volume and searches for unreliable (bad) storage areas.
Extended Directory	Lists complete directory information about a specified volume or set of files.
Krunch	Consolidates all unused space on a volume in a single area by packing the existing files together. (Not valid for SRM or HFS)
List Directory	Lists partial directory information about a specified volume or set of files.
Prefix	Specifies a new default volume.
Volumes	Lists the volumes currently on-line.
Udir	Sets the default unit directory. (HFS and SRM only)
Zero	Creates an empty directory on the specified volume. (Not valid for HFS or SRM)

Exit Commands

Quit	Provides an orderly exit from the filer.
Stop	Pressing the Stop key exits the Filer Subsystem unconditionally. The current I/O operation is completed before exiting.

File Related Commands

Access	Change the access rights (passwords) on a file or directory. (SRM only)
Change	Change the name of a file, set of files, or volume.
Duplicate link	Duplicates links to a file or set of files. (HFS and SRM only)
Filecopy	Copies a file, set of files, or a volume to a specified destination.
Hfs	Change the access rights (modes) and owners of files and directories on an HFS disc (HFS only).
Make	Create a directory (HFS and SRM) or a file on a volume.
Remove	Remove a directory entry or a set of directory entries.
Translate	Translates text files of types TEXT, ASCII, UX and Data to other text file representations or to un-blocked volumes.

Workfile Related Commands

Get	Specifies a file as the workfile.
New	Specifies that no file is the current workfile.
Save	Saves the current workfile(s) with the specified name.
What	Lists the name and current state (saved or not saved) of the workfile(s).

Librarian Command Summary

General Commands

Boot	Creates "system Boot files."
Edit	Gets you into Edit mode, for either Copying or Linking.
File	Sends the File directory (all module names) to the current Printout file.
Header	Allows you to specify the Header size for the Output file.
Input	Allows you to specify the Input file.
Keep	Makes a permanent copy of the current Output file.
Output	Allows you to specify the Output file.
Printout	Turns the Printout option ON or OFF, or allows you to specify a Printout file.
Quit	Quits the Librarian and returns you to the Main Command Level.
Unassemble	Gets you into the Unassemble mode.
Verify	Gets you into the Verify mode, and shows the name of the first module in the Input.

Copy Mode Commands

All	Transfers All modules from the Input file to the Output file.
Link	Gets you into Link mode.
Module	Allows you to specify the next Module to be copied from the Input file.
Transfer	Transfers the current object module to the Output file.

Edit Mode Commands

Append	Used to Append modules to the Output file.
Copy	Copies the First module up to (but not including) the Until module to the Output file.
First	Allows you to specify the First module to be transferred to the Output file. (The First module must precede the Until module in the Input file.)
Stop	Stops the Edit session and returns to the Librarian's main prompt.
Transfer	Transfers the current object module to the Output file.
Until	Allows you to specify the Until module.

Link Mode Commands

All	Transfers All modules from the Input file to the Output file.
Copy	Returns you to Copy mode.
Def	Controls whether or not the DEF table is included in the Output file.
Global	Allows you to change the Global base address of the module.
Link	Finishes Linking.
Module	Allows you to specify the next Module to be copied from the Input file.
New	Allows you to name the New object module being created.
Relocation	Designate the Relocation base address to be used.
Space	Assigns Space for patches.
Transfer	Transfers the current object module to the Output file.
X	Allows you to enter a copyright notice as part of the Output file.

Unassemble Commands

Assembler	Directs the Librarian to unassemble the Input file using Assembler conventions.
Compiler	Unassembles all lines of the Input file according to Compiler conventions.
Def	Sends the DEF table to the Printout file.
Ext	Sends the EXT table to the Printout file.
Line range	Unassemble (using Compiler conventions) a section of code defined by two Line values.
PC range	Unassemble (using Assembler conventions) a section of code defined by two location counter range values.
Stop	Stops the unassemble session and returns to the Librarian's main prompt.
Text	Sends the interface Text (DEFINE SOURCE) of the current Input module to the Printout file.

Debugger Command Summary

Register Operations

A0..A7, **D0..D7,** **PC, US, SR**	Display or assign values to the processor registers.

Breakpoint Commands

BS	Set a breakpoint at the specified location.
BD	Disable (but don't remove) breakpoints.
BA	Activate disabled breakpoints.
BC	Clear and remove breakpoints.
B	Display the breakpoint table.

Call Command

CALL	Calls the machine-language routine at the specified address.

Display Command

D	Display the specified object(s) immediately, directly, or indirectly.

Dump Commands

DA	Performs a DUMP ALPHA function.
DG	Performs a DUMP ALPHA function.

Escape Commands

EC	Generates the specified escape code.
ET	Specify escape codes to be trapped by the Debugger.
ETC	Sets up trapping of all escape codes; the Debugger executes specified command(s) when an escape is encountered.
ETN	Specify that all escape codes except the ones listed are to be trapped by the Debugger.

Format Commands

FB	Sets the default display format to Binary.
FH	Sets default format to hex values.
FI	Sets default format to signed integer values.
FO	Sets the default display format to Octal.
FU	Sets default format to unsigned integer values.

Go Commands

G	Resume execution.
GT	Resume execution until a specified location is reached. This is a BS and G combined.
GTF or **GFT**	Same as GT except execution is slowed and the line numbers are flashed in the lower right corner of the CRT.

IF, ELSE, and END Commands

IF, ELSE, END	Allow conditional execution of Debugger command(s).

Open Memory Commands

OB, OW, OL	Display or alter memory locations.

Procedure Commands

PN	Continues the program, but halts program execution when the next procedure is called (or current one is exited, whichever occurs first).
PX, or **P**	Continues the program, but halts program execution when the current procedure is exited.

Queue Commands

Q	List the most recent line numbers (or PC values if Trace commands were used with machine code).
QE	Terminate recording of line number values in the queue.
QS	Start recording the information in the queue.

Softkey Commands

k0...k9	Define softkeys as typing-aid keys.

System Boot Commands

sb	The system boot command puts the computer in the Boot ROM to cause a boot/reboot operation.

Trace Commands

T.	Execute the specified number of instructions, each followed by a TD command.
TD	Display the command string defined by the softkey **k4**.
TD I	Restores the initial command string to **k4**.
TQ	Same as the T command except the TD command is executed only after the last instruction.
TT	Same as the TQ command except a location is specified rather than a count.

Walk Procedure Links Commands

WD	Move the stack frame pointer to the stack frame of the calling procedure.
WS	Move the stack frame pointer to the stack frame of the nesting procedure.
WR	Return the stack frame pointer to the current stack frame.

Glossary

E

ASCII character Any of the 8-bit characters in Hewlett-Packard's extended ASCII (American Standard Code for Information Interchange) set. The characters include letters, numerals, punctuation, control characters, and foreign character sets. A table of these characters and their code values can be found at the end of this glossary.

absolute addressing Using the actual 32-bit address of a variable or entry point to specify its location.

anchor An internal pointer used by the Editor's Zap command as a starting point for removing text. The anchor is set at the cursor position of the most recent Adjust, Find, Insert or Replace command. The cursor is moved to the anchor location by the Equals command.

The **ANY CHAR** key can be used to generate characters which may not otherwise be obtainable by regular keystrokes. To use it, press **ANY CHAR** and then type in any three digits. If the number entered is larger than 255, the system divides it by 256 and uses the remainder as if it were the number. The character generated corresponds to an integer in the range 0 thru 255. For example, the character P is ASCII character number 048. You need to use all three digits (including leading zeros) with the **ANY CHAR** key to get the right results. The integer values and their corresponding characters in HP's extended ASCII set are shown in the ASCII table at the end of this glossary.

auto indent One of the Editor's environment parameters, this true/false switch affects the Insert and Margin commands. When Inserting text, pressing Return (Enter) causes the cursor to be positioned at the same starting column as the previous line if auto indent is true. If auto indent is false, the cursor is positioned at the left margin as defined in the Editor's environment. When this switch is true, the Margin command is disabled and Filling is not done while indenting text.

bit An abbreviation for the term "binary digit", a bit is a single digit in base 2 that must be either a 0 or a 1.

block A block is a 512 byte unit of storage area on a WS1.0 volume and a 256 byte sector on a LIF volume. The Pascal system allocates storage space for files on the WS1.0 and LIF volumes in block increments. HFS blocks (fragments) are usually 1024 bytes; a file's storage space is allocated in block increments usually.

block-structured An attribute of a device which structures its memory allocation in block units such as flexible or hard discs. Devices such as printers and screens (CRTs) are not block-structured.

breakpoint Used in the Debugger subsystem, this is a location (either an address or a line number in a Pascal program) where program execution is interrupted. Often a breakpoint has an operation associated with it. This operation is performed when the breakpoint is encountered.

boot device The peripheral where the Boot ROM found and loaded the Pascal operating system. The Boot ROM has a search pattern which allows booting from just about any drive in any HP mass storage product, including the Shared Resource Manager.

bus address When several peripherals are connected to the same HP-IB interface, a bus address is required (in addition to the select code) to designate the particular peripheral referenced by an I/O transaction.

byte A group of eight bits processed as a unit.

command character One of the environment's parameters in the Editor, this character functions as a delimiter for paragraphs. It is often used to protect text from being accidentally Margined. The Command character can be any non-control ASCII character. The default command character is the caret (^).

control character Any ASCII character whose value is either 127 or in the range of 0 thru 31. Use of control characters in the Editor and Filer is discouraged as they may have undesirable effects.

copy buffer A temporary storage buffer used by the Editor's Copy command and filled by the text involved in the most recent Delete, Insert or Zap command. Copying from a file clears the contents of the copy buffer and the Margin command clears the copy buffer regardless of the environment settings.

cursor The flashing underline (_) symbol on the screen. The cursor functions as a reference point for Editor commands which manipulate text and as a reference for prompts in other Pascal subsystems.

cursor control keys Keys which control the movement of the cursor on the screen. These are the four arrow keys, the [Tab] and [Return] ([Enter]) keys, the space bar, and the cursor wheel.

cursor wheel The wheel (also called the knob) on the upper left area of the keyboard whose action duplicates that of the four arrow keys. When used in the Editor with the [Shift] key pressed, turning the wheel produces up or down cursor movement; unshifted, it produces left or right cursor movement.

DEF table (definition symbol table) There is only one DEF table per object module. It contains one DEF record for each symbol which is exported from the module. Each DEF record has two parts. The first part is a packed string containing the name of the symbol which is defined. 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.

define source There may be one block of define source per object module. It begins on a block boundary, which is given in the module directory along with the length. The define source may be any arbitrary text, but it is intended to be a copy of the define section from a Pascal module. It is this section of the module which is accessed when it is imported or used by the compiler. The define section of a Pascal module contains the reserved words `MODULE`, `IMPORT` and `EXPORT` plus all declarations made with these words.

delimiter In the Editor, a delimiter is any item which defines the beginning and ending of either a string or a paragraph. Delimiters used to define strings can be any of the following characters:

```
! " # $ _ ' ( ) * + , - . / ; < = > ? @ [ \ ] & ^ { | } ~
```

Delimiters must be used in matching pairs. Paragraph delimiters are any combination of blank lines, lines whose first non-blank character is the Command character, or the beginning or end of a file.

device specifier By convention in the Pascal, BASIC and HPL systems, when select code and bus address are used together to address a peripheral, they are concatenated into a single number. Thus the device at address 1 on select code seven is referenced as "701", which is derived by multiplying the select code by 100 and adding the address. **NOTE** some HP products contain, within a single package, several peripheral devices which are addressed separately.

direction In the Editor, defines how cursor movement (generated by [Tab], [Return] [Enter] and the space bar) and string searches (made by Find and Replace) occur in the text file. The first character of all command prompts in the Editor (except Quit) shows the current direction. If forward (>), cursor movement and searches take place from the cursor position toward the end of the file. If backward (<), these actions occur between the cursor position and the beginning of the file. Direction is set forward by pressing [>] [.] or [+] and set backward by pressing [<] [,] or [-].

directory Contains information about the files on a volume. This information includes the volume name and the following information about each file on the medium: the file name, the file size (in number of blocks), the date of last modification to the file, its starting block address, and the file type (which reflects the file's attributes). Directory information can be seen by using the Filer's List Directory and Extended Directory commands. The directory is initialized with the Filer's Zero command (except SRM and HFS).

Directory Access Method or DAM Each mass storage unit has a directory describing the files it contains, the type of each file and so forth. Many different directory organizations are used within HP, and data on a disc can't be interpreted properly unless it is accessed using the correct Directory Access Method. Pascal 3.2 supports four DAMs: the "Workstation" format compatible with Pascal 1.0 systems; HP's "Logical Interchange Format" or LIF directory; the Shared Resource Manager's hierarchical directory; and HFS, which is compatible with Series 200 and 300 HP-UX revision 5.1.

document environment A predefined Editor environment configuration suitable for writing and editing non-program text.

dollar sign This character "$" is used in the Filer as a convenience in specifying file names. When used in place of a destination file name, it means that the file is to have the same name as the source file.

Editor prompt This is displayed on the top line of the screen when the Editor is expecting a command. Its first character displays the current direction. The rest of the prompt shows the most common Editor commands in abbreviated form followed by a question mark. Typing [?] displays a second prompt with the rest of the commands and with the Editor's revision number shown in brackets.

The [Enter] key generates a carriage return and is used to end lines when responding to a request for information by one of the Pascal subsystems. Pressing [Enter] alone in response to a command prompt while in the Filer exits the command and returns the Filer's prompt. [Enter] is also used to end lines when Inserting text in the Editor.

entry point The place where a program or subroutine begins. Before a routine is executed, the address of the entry point must be obtained from a symbol table and that address is put in the program counter.

environment The conditions or parameters which affect how text in the Editor is Adjusted, Inserted, and Margined. These parameters may be changed with the Editor's Set command.

exclamation point The Editor shows this character (!) in the rightmost column of the screen whenever a line of text goes beyond the right boundary of the display area. The text remains intact in the computer's memory. Inserting a carriage return with the [Return] ([Enter]) key, redefining the environment with the Set command or using the Adjust command to shift the text to the left will generally remedy the situation.

EXPORT Export is a reserved word used in a Pascal module. It is used to name those procedures, functions, constants and variables that are exported or made available to importing modules.

external reference A reference to a symbol outside of a program. If the address of a symbol cannot be found in the program symbol table, a search is made of the system symbol table.

EXT table (external symbol table) There may be one EXT table per object module. The EXT table contains one EXT record for each external symbol. An external symbol is one that is used in a module but not located in the module. Before a module can be executed, the location of all external symbols must be obtained from the REF tables of other modules and associated with each external symbol in the module.

file A discrete collection of information designated by a file name and residing on a mass storage medium.

file name An entry in a directory which identifies a particular file.

file specification Completely identifies a file and includes both a volume specification and a file name. A volume specification can be one of many items, but it is always part of a file specification. If a volume ID is given, it must be separated from the file name by a colon (:). If not, the default volume is assumed.

file types Several file types are recognized by the Pascal System. Files generally (but not always) have a suffix as part of the file name which indicates their type. The file type is established at the time of the file's creation and cannot be changed just by changing the suffix. The types and their associated suffixes are:

TEXT files — (suffix is .TEXT) Contain ASCII characters and Editor environment information.

ASCII files — (suffix is .ASC) Are similar to TEXT files. The format is slightly different and there is no Editor environment information.

UX files — (suffix is .UX) Are similar to Data files. Format is compatible with HP-UX text files and contain no Editor environment information. May contain non-text data.

CODE files — (suffix is .CODE) Contain code generated by the Pascal Assembler, Compiler or Librarian.

Data files — (no specific suffix) Are files which can be created by any subsystem but are used primarily as INPUT and OUTPUT files in Pascal programs. They do not have suffixes.

System files — (suffix is .SYSTM) Are files created with the Librarian's Boot command. They are loadable by the boot ROM.

Bad files — (suffix is .BAD) are a type of file created by the user to isolate unreliable or worn-out areas on a mass storage medium. Once created, BAD files will not be moved by subsequent crunches of the volume.

Filer prompt This is displayed on the top line of the screen when the Filer is expecting a command. The line shows the most common Filer commands in abbreviated form followed by a question mark. Typing [?] displays a second prompt that shows the rest of the commands and the Filer's revision number enclosed in brackets.

filling A true/false parameter in the Editor's environment which affects the Insert and Margin commands. Filling must be set true (and Auto-indent false) while Inserting to cause text to "wrap around" to the next line. If true when an Insert is confirmed with [Select] (**EXECUTE**), Filling causes the rest of the paragraph to be Margined according to the left, right and paragraph margin values of the environment. If Filling is false while Inserting text, the Editor generates a beep as text approaches the right edge of the screen. Filling must be true (and auto indent false) for the Editor's Margin command to work.

IMPLEMENT A reserved word used in a Pascal Module. It is used as a flag to indicate the beginning of the module body. It consists of the reserved word plus a declaration statement. The statement can be either empty or used to declare those constants, variables, procedures and functions used internally by the module. None of this information is available outside the module.

IMPORT A reserved word used in a Pascal Module. It is used to name those object modules imported or linked to the importing file at execution time. In a program, IMPORT names the modules upon which the program depends.

inode Each file or directory on an HFS disc has an inode which contains information about the file's size, mode, number of links, location on the disc, dates, etc.

interface The electronic circuitry which connects the computer's high-speed internal bus to lower speed physical peripheral devices. Interfaces are either built-in, like the standard HP-IB port at the back of your computer, or plug into the I/O backplane. Most of the peripherals supported by the Series 200/300 computers are designed to connect through an HP-IB interface.

interface text The IMPORT and EXPORT information from a module which must be known by the Compiler in order to combine it with other modules.

knob The rotary-pulse generator that is used as an input device on the built-in keyboards of the 9826 and 9836, and on optional HP 98203B or 98203C keyboards. It is used in the Editor for moving the cursor (in that context, it is referred to as the "cursor wheel"). You can use it in programs for any purpose that you like.

Librarian A Pascal subsystem designed to manage object modules. It can link or just collect object modules together into object files. The Librarian is the file named LIBRARIAN in the operating system and is accessed by pressing [L] from the Main Command Level.

LIBRARY A special library included with the Pascal operating system. The LIBRARY file should be kept on-line so that object modules stored in it are automatically available to any program importing them.

literal In the Editor, a search option used for a target string in the Find and Replace commands. A literal is any occurrence of a string. This can be an isolated string or one embedded in either a word or paragraph. (See Token).

Main Command Level The level from which all the subsystems of the Pascal System are entered. The prompt displayed at this level looks like:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

margins (left, right and paragraph) Parameters in the Editor's environment which affect the Adjust, Insert and Margin commands. Adjust uses these to move text to the Left margin, Right margin or to center text between the two. Inserted text falls within all three defined margins if Auto indent is false and Filling is true. Margin causes all text in the paragraph where the cursor is located to conform to the three margin settings.

marker Used in the Editor's Copy, Jump and Set commands, a marker is a pointer in a text file whose location is associated with a name. A marker name can be any sequence of up to eight non-control ASCII characters. The Editor truncates anything over eight characters and converts all lowercase letters to uppercase. Ten markers are allowed in each text file.

module See "object module".

mouse A small, rodent-like input device, consisting of a roller ball and buttons. Rolling the device on any surface generates two-dimensional movement information that is transmitted through its tail to the computer. Pushing the buttons also generates information that is sent to the computer. The mice available with Series 200 and 300 equipment are connected to the computer through the HP Human-Interface Link (HP-HIL).

object file An object file is a unit of binary code managed by the Librarian. It is made up of a Library directory and one or more object modules. The Assembler and Compiler generate one object file per source file. The Compiler's object file can contain one or more object modules depending upon the source file's construction. If the source file contains a number of compilable modules, that number of object modules will be created in the object file.

object module Contains the interface information necessary to link and run the module and the machine code.

on-line Any object (device, volume or file) currently accessible by the Pascal System.

opcode A word that stands for one of the operations of the microprocessor or coprocessor. The Assembler translates these words into actual binary codes which the microprocessor or coprocessor understands.

operand The symbol which stands for the object on which microprocessor operations are performed.

page In the Editor, one full screen display (23 lines).

paragraph A paragraph is text in the Editor delimited by blank lines or the beginning or end of a file. Margin, as well as the Insert command (if confirmed by pressing [Select] [**EXECUTE**]), both use this definition when Margining text.

Pascal module HP Pascal allows program modules to be compiled separately into object modules. The modules are generally not executable, but are parts of Pascal programs. The sections of a module are:

MODULE
IMPORT
EXPORT
IMPLEMENT

pass by reference The address of a parameter variable is given to the called routine. Using that address, the routine can alter the value of the variable.

pass by value The current value of a variable is given to the called routine. In this way the value can be used but the routine does not alter the actual variable.

peripheral An I/O device such as a printer or disc. Devices such as plotters and digitizing tablets are also peripherals, but they are accessed through the I/O library rather than the Pascal file system. For the present discussion we use the term to refer only to devices accessible through File System operations.

program environment A predefined Editor environment configuration suitable for writing and editing program text. Also, the environment chosen as the default environment when the Editor is entered with neither a workfile nor a specified file.

prompt Generally, any request for information from the system. The different Pascal subsystems have primary prompts (the Editor Prompt, Filer Prompt, etc.) and many subsystem commands have prompts of their own which are displayed at the top of the screen when the command is entered.

pseudo-op A word which stands for an operation which the Assembler performs rather than an operation which the microprocessor performs.

REF table Is used in resolving both external references and internal references that cannot be resolved using PC relative addressing. For more information, consult the System Designer's Guide.

relative addressing An addressing mode where the location of a routine or variable is given as an offset from the current location rather than an absolute address. In this way, the code can be placed at different places in memory without having to change the addresses of variables and entry points.

repeat factor An Editor option used to repeat the effects of any cursor control key and used in the Find, Page and Replace commands. A repeat factor is generally a positive integer in the range 1 to 9999. For [Tab], the upper limit is 4095; for the Page command, 1000.

The [Return] key generates a carriage return and is used to end lines when responding to a request for information by one of the Pascal subsystems. Pressing [Return] alone in response to a command prompt while in the Filer exits the command and returns the Filer's prompt. [Return] is also used to end lines when Inserting text in the Editor.

same An Editor option in the Find and Replace commands which allows you to chose a previously used string instead of having to specify a string surrounded by delimiters. Just type and "s" in place of the string.

select code A number between 0 and 31, the "address" or name by which an interface is identified and referenced. When a peripheral operation is performed, it takes place through an interface which is said to be "on a select code". Most interface cards which plug into the I/O backplane have switches which can be set to indicate the select code to which the interface will respond. The built-in interfaces have fixed select codes.

size An optional parameter used in the Filer's Make and Transfer commands to specify the number of blocks in a file.

slash character A "/" character used in the Editor which has the same effect as using a very large repeat factor. The slash can be used with any cursor control key and with the Adjust, Find, Page and Replace commands.

The [Stop] key when pressed, is used to leave any of the Pascal subsystems and return to the Main Command Level. When used from the Editor, you are asked if you really mean to leave without saving the current file.

string A contiguous series of characters.

structured constant A constant that has more than a single value, such as a record or array.

structured variable A variable that has more than a single value, such as a record or array.

substitute string Specified in the Editor's Replace command, this string takes the place of the target string in the text file. The string can be empty (null) and of a different size than the target string. The "same option" can be used in place of specifying a string surrounded by delimiters.

symbol table A table containing the address locations of the variables and routine entry points.

system volume or system unit The Pascal system distinguishes one mass storage unit to be used for special purposes. This "system volume" is where the date and any AUTOSTART file are found at boot time. It is where the system looks first for system files such as the Compiler and Editor, where workfiles are stored, and where an intermediate file is stored during interpretation of a Stream (command) file.

target string When specified in the Editor's Find command, the cursor moves to the specified string. When used in the Replace command, the target string is the one replaced by the substitute string. The "same option" uses the most recent target string, regardless of whether it was used by the Find or Replace commands. The maximum length of a target string is 128 characters.

text file A file created and/or used by the Editor which contains ASCII or selected foreign characters. The Editor automatically appends `.TEXT` to a file name unless it either already contains a suffix or the last character in the file name is a period. A text file may be of type TEXT, ASCII, DATA, or HP-UX compatible.

token An Editor search option used for a target string in the Find and Replace commands. A token is an isolated string delimited by any two ASCII characters which are neither letters nor numerals. The delimiters do not have to match each other (i.e., the token: **`again.`** is delimited by a blank and a period). (See Literal).

unit An entry in the Unit Table.

unit table The Pascal system provides for up to 50 units, designated #1 through #50. They are represented by a 50-entry array called the Unit Table or "Unitable". Each entry fully specifies the association of one logical unit to a physical peripheral, with such information as the device specifier and driver procedures to be used for I/O operations to the unit.

unit number An integer in the range from 1 through 50 representing the volume having the corresponding entry in the unit table.

verify An option which allows you to confirm the substitution of one string for another in the Editor's Replace command.

volume A volume refers to any I/O device such as a printer, keyboard, screen, or mass storage device. The name of a mass storage volume is found in its directory; the name of an unblocked device is found in its Unit Table entry. There may be several volumes on one physical storage medium. Hard discs typically contain multiple volumes, but flexible discs generally have only a single volume. The volume may be mounted (in a disc drive) or not. The syntax of a volume name depends on its type (for example, LIF volume names may contain 6 characters, WS1.0 may contain 7, SRM may contain 16, and HFS may contain 6 at the root and 14 elsewhere).

wildcard Both of the characters = and ? can be used in the Filer as wildcards in place of parts of a file specification.

workfile If the workfile exists, it is the automatic file used by the Editor, Compiler, Assembler, Debugger and the Run command. It is designated when quitting the Editor using the Update option or the Filer's Get command.

Utility 20-2
Verification 20-10
Bad sector command (Filer) 5-29
Blocked and Unblocked Units 3-3
Blocked devices 18-10
Boot files (Renaming) 18-35
Boot ROM 18-4
Boot volume (defined) 18-6
Boot-time errors A-2
Booting from EPROM 19-20
Booting process 18-4
Breakpoints (Debugger) 9-13
BRSTUFF module (CTABLE) 18-63
Bubble memory:
Configuration 19-3
Driver module 19-5
Error correction 19-10
File System access of 19-10
Hardware device 19-11
Initializing 19-11
Interrupts 19-11
Introduction 19-1
Using 18-19, 19-3

C

CALL 12-4
CALLABS (Compiler option) 6-26
Cartridge tape drives 19-35
Change command (Filer) 5-18, 5-30
Changing memory contents (Debugger) 9-22
Chapter previews 2, 4
Character sets C-1
Clock 22-4
CLOSE (files) 15-25
Coalescing hard-disc volumes 18-14, 18-26
CODE (Compiler option) 6-27
Code file specification 9-6
CODE_OFFSETS (Compiler option) 6-28
Command Interpreter 18-6
Command reference:
Debugger 9-37
Librarian 8-24
Command summary:
Debugger 9-34, D-8
Editor D-2
Filer D-4
Librarian D-6
Main Command Level D-1
Commands:
eXecute (Main level) 2-5

Initialize (Main Level) 2-6
Main Level 2-3, 2-4
Memory volume (Main Level) 2-7
New sysvol (Main Level) 2-9
Permanent (Main Level) 2-10
Run (Main Level) 2-11
Stream (Main Level) 2-12
Syntax diagram 2-4
User restart (Main Level) 2-15
Version (Main Level) 2-16
What (Main Level) 2-18
Compatibility hardware 22-10
Compiler option:
ALIAS 6-23
ALLOW_PACKED 6-24
ANSI 6-25
CALLABS 6-26
CODE 6-27
CODE_OFFSETS 6-28
COPYRIGHT 6-29
DEBUG 6-30
DEBUG ON 6-55
DEF 6-31
FLOAT_HDW 6-32
General 6-22
HEAP_DISPOSE 6-33
IF 6-34
INCLUDE 6-35
IOCHECK 6-36
LINENUM 6-37
LINES 6-38
LIST 6-39
OVFLCHECK 6-40
PAGE 6-41
PAGEWIDTH 6-42
PARTIAL_EVAL 6-43
RANGE 6-44
REF 6-45
SAVE_CONST 6-46
SEARCH 6-47
SEARCH_SIZE 6-48
STACKCHECK 6-49
STACKCHECK_ON 6-56
SWITCH_STRPOS 6-50
SYSPROG 6-51, 17-1
TABLES 6-52
UCSD 6-53
WARN 6-54
Compiler:
Absolute address (of variables) 11-6

Absolute addressing (of variables) 11-6
ANYPTR type 11-6
CALL 12-4
Error trapping and simulation 17-1
Errors 6-19, 6-21
errors A-9
Function calls 6-58
Function results 6-59
Global variables 6-56
INCLUDE files 6-18
Introduction 6-1
Invoking 6-3
IOCHECK 17-3
IORESULT function 17-2
Listing 6-5
Mixing DISPOSE and RELEASE 16-4
Modules 6-7
Parameter passing 6-58
Procedure calls 6-56
Relaxed typechecking 12-2
Running the program 6-5
SEARCH option 6-14
Separate module compilation 6-13
Stack usage 6-55
Static links 6-59
Strategy for compiling modules 8-5
UCSD options 6-53
Variable size 11-5
Workfile 6-6
Compiling modules 8-5
Configuration:
Example of SRM 18-72
Interfaces 18-16
Modifying the standard 18-26
Multi-disc SRM 18-83
Verifying changes to 18-69
Copy command (Editor) 4-30
Copying discs 5-13
Copying files 5-16
Copying files (to SRM) 18-78
Copying system files 18-34
COPYRIGHT (Compiler option) 6-29
Creating an HFS directory 5-14
Creating an SRM directory 5-14
CRT highlight characters C-12
CS80 discs (configuration) 18-26
CTABLE program:
BRSTUFF module 18-63
Commentary 18-51
Compiling 18-69

CTR module 18-62
Device address vectors 18-54
Editing 18-68
Flexible disc units 18-53
Local printers 18-53
Modifying 18-50
Modifying for Bubble cards 19-8
OPTIONS module 18-52
Running 18-69
SCANSTUFF module 18-63
Secondary DAMs 18-52
CTR module (CTABLE) 18-62
Cursor wheel 4-8, 4-21

d

DAMs (Directory access methods) 2-7, 18-23, 18-52
Data types:
ANYPTR 12-3
ANYVAR 12-2
Data-Cartridge tape drives 19-35
DEBUG (Compiler option) 6-30
DEBUG ON (Compiler option) 6-55, 9-2
Debugger commands:
B 9-39
BA 9-39
BC 9-40
BD 9-40
BS 9-41
CALL 9-42
D 9-43
DA 9-45
DG 9-45
EC 9-46
ET 9-46
ETC 9-47
ETN 9-47
FB 9-48
FH 9-48
FI 9-48
FO 9-48
FU 9-48
G 9-49
GF 9-49
GT 9-50
GTF 9-50
IF, ELSE, END 9-51
OL,OW,OB 9-53
PN 9-54
PX 9-54
Q 9-55

QE 9-55
QS 9-55
Register operations 9-56
sb (system boot) 9-58
Softkey commands 9-57
T 9-58
TD 9-59
TQ 9-59
TT 9-59
WD 9-60
WR 9-60
WS 9-60
Debugger:
Breakpoint Table 9-15
Breakpoints 9-13
Changing memory contents 9-22
Clearing Breakpoints 9-14
Code file specification 9-6
Command reference 9-37
Command screen 9-7
Command summary 9-34, D-8
DEBUG Compiler option 9-2
Default display formats 9-12
Display formats 9-10
Displaying data 9-9
Errors A-14
Examining consecutive memory 9-20
Examining variables 9-18
Example program 9-2
Exception trapping 9-24
Executing a number of statements 9-16
Expressions 9-37
Formats for structured variables 9-21
Generating Escapes 9-25
Input formats 9-12
Introduction 9-1
Invoking 9-6
Is it installed? 9-5, 9-31
Key notation 9-5
Keyboard 9-30
Loading 9-1, 9-4
Named Reboot 9-27
Pause function 9-16
Prompt 9-7
Queue 9-9
Sample session 9-2
Screen dumps 9-8
Single-stepping 9-7
Slow program execution 9-7
Stack frame 9-17

Static and dynamic links 9-23
Tracing program flow 9-17
DEF (Compiler option) 6-31
DEF table 7-7, 8-27
DEF table command (Librarian) 8-18
Default display formats (Debugger) 9-12
Default volume 2-18, 3-4, 5-3
Define Source 8-28
Delete command (Editor) 4-32
Deleting files 5-19
Device address vectors (CTABLE) 18-54
Device classes (TABLE program) 18-10
Device drivers 18-38
Device priority (while booting) 18-11
Device-driver modules 18-7
Direct access files 15-28
Directory access methods (DAMs) 2-7, 3-10, 18-21
Directory path syntax 3-6
Disassembly of a module 7-43
Disc drives 3-2
Disc interleave 15-41
Disc performance 18-18
Discs (general) 3-1
Discs (system) 18-5
Display formats (Debugger) 9-10
Displaying data (with Debugger) 9-9
Displays (Series 200/300) 22-2
DISPOSE 16-3
DMA card (configuration) 18-3
Drive numbers 3-3
Duplicate command (Filer) 5-32

e

Editor command:
Adjust 4-28
Copy 4-30
Delete 4-32
Equals (=) 4-34
Exchange 4-53
Find 4-35
Insert 4-37
Jump 4-40
Margin 4-41
Page 4-42
Quit 4-43
Replace 4-45
Set 4-48
Verify 4-52
Zap 4-55
Editor:

Anchor 4-21
Backing up your file 4-20
Changing text 4-11
Command summary D-2
Confirming or aborting commands 4-7
Copying text from other files 4-6
Creating a text file 4-3
Creating text 4-4
Cursor 4-21
Deleting text 4-9
Duplicating text 4-11
Entering the 4-2
Exiting the Editor 4-19
File size 4-22
Finding text patterns 4-11
Formatting text 4-16
Introduction 4-1
I/O errors 4-24
Margining text 4-16
Moving text 4-11
Moving the cursor 4-8
Recovering deleted text 4-11
Setting the environment 4-19
Storing your file 4-5, 4-19
Stream files 4-24
Text file structure 4-23
Using workfiles 4-24
Window 4-21
EPROM memory:
As the system volume 19-20
Blank check 19-27
Burn failure 19-29
Burn rate 19-25
Check failure 19-29
Configuration changes 19-13
Configuration modifications 19-32
Driver module 19-30
Driver modules 19-13
Empty sockets 19-26
File system access 19-34
Headers 19-20
Introduction 19-1
Memory addresses 19-17
Memory card installation 19-16
Overview of using 19-12
Programmer card installation 19-14
Programmer card select code 19-14, 19-25
Programming utility 19-19
Transfer utility 19-23
Transferring files 19-21

Transferring volumes 19-20, 19-28
Using 18-19, 19-12
Equals (=) command (Editor) 4-34
Errors:
Assembler A-13
Boot-time A-2
Compiler A-9
Debugger A-14
Graphics errors A-7
I/O library A-6
I/O system A-4
Loader/Segmenter A-8
Messages A-1
Recovery 7-13
Run-time A-3
Syntax 6-4
Trapping and simulation 17-1
VMELIBRARY A-16
Examining consecutive memory (Debugger) 9-20
Examining variables (Debugger) 9-18
Exception coding 7-15
Exception trapping (Debugger) 9-24
eXchange command (Editor) 4-53
eXecute Command (Main Level) 2-5
EXPORT 8-29
Expressions (Debugger) 9-37
EXT Table 7-8, 8-28
EXT table command (Librarian) 8-18
Extended directory command (Filer) 5-7, 5-34
Extensions (to Pascal) 10-10
EXTERNAL procedures 7-16

f

Failure of TABLE program 18-13
File directory 8-2
File specification 3-6
File System (Introduction) 3-1
File types 3-13
Filecopy command (Filer) 5-13, 5-16, 5-37
Filer:
Access command 5-26
Bad sector command 5-29
Change command 5-18, 5-30
Command summary D-4
Confirming or aborting commands 5-2
Creating a directory (HFS) 5-14
Creating a directory (SRM) 5-14
Deleting files 5-19
Duplicate command 5-32
Duplicate_link 18-78

Entering the Filer 5-2
Extended directory command 5-7, 5-34
Filecopy command 5-13, 5-16, 5-37
Get command 5-40
HFS access rights 5-8
Hfs command 5-41
Introduction 5-1
Krunch command 5-44
Leaving the Filer 5-20
List-directory command 5-6, 5-46
Make command 5-48
New command 5-50
Prefix command 3-5, 5-51
Prompt 5-2
Quit command 5-53
Remove command 5-54
Removing files 5-19
Save command 5-56
SRM access rights 5-8
Stream command 2-12
Translate command 3-15, 5-11, 5-57
Unit directory command 5-60
Volume back-up 5-13
Volumes command 3-2, 5-61
What command 5-62
What devices are accessible? 5-3
Wildcards 5-10
Workfile 5-20
Zero command 5-63
Files:
APPEND 15-24, 15-25, 15-34
Buffer Variable 15-22
CLOSE 15-25
Creating a text file 4-3
Current component 15-22
Debugging 15-40
Declaring a TEXT file 15-30
Deleting 5-19
Disposing of 15-25
F^ 15-22
File buffer 15-22
File modes 15-22
File position 15-22
File specification (syntax) 5-24
File variable 15-6
Formatted I/O 15-33
General discussion 3-5
GET 15-27
HFS names 3-12
HFS permissions 15-40

Interchange between BASIC and Pascal B-18
LIF file names 3-10
Lookahead mode 15-23
MAXPOS 15-30
Names to avoid 3-9
Naming conventions 3-6
Object (definition of) 8-2
Object modules 6-1
OPEN 15-24, 15-25, 15-34
Opening existing 15-25
Pascal operations 15-20
POSITION 15-30
Programming with 15-1
PUT 15-27
READ 15-26
Read mode 15-22
READDIR 15-28
Removing 5-19
Renaming 5-18
RESET 15-25, 15-34
REWRITE 15-5, 15-24, 15-26, 15-34
SEEK 15-29
Sequential operations 15-26
Size specification 3-12, 15-21
Specification 3-6
SRM concurrent file access 3-24, 15-37
SRM names 3-11
Stream files 2-12, 4-24
Structure of text files 4-23
Suffixes 3-13
Suppressing the suffix 3-15
Syntax of name 3-6
Systm 18-4, 18-6
Temporary 15-21
Text file representation 15-31
Textfile I/O 15-30
Translating between data types 3-15
Types 3-10, 3-13, 15-4
Wildcards 3-16, 5-10
Workfile 4-24, 5-20, 6-6
WRITE 15-27
Write mode 15-23
WRITEDIR 15-29
WS1.0 file names 3-18
Find command (Editor) 4-35
Flags 8-29
Flexible discs (CTABLE) 18-53
FLOAT_HDW (Compiler option) 6-32
Floppy drives (in the Unit Table) 18-11
Formats for structured variables (Debugger) 9-21

Formatted I/O (files) 15-33
Full backup 20-3
Function calls 6-59
Function results 6-59

g

General Value or Address Record (GVR) 8-30
Generating Escapes (Debugger) 9-25
Get command (Filer) 5-40
GET (files) 15-27
Global base 6-56
Global space 2-16
Global variables 2-10, 2-15, 6-56, 7-9
Glossary E-1
Glossary (Librarian) 8-27
Graphics errors A-7
Graphics input and output 18-17

h

Hard disc:
  Partitioning 18-11, 18-14, 18-55
  Unit Numbers 18-11
  Volumes 18-14, 18-26
Heap Management:
  DISPOSE 16-3
  MARK 16-2
  RELEASE 16-2
HEAP_DISPOSE (compiler option) 6-33
Hfs command (Filer) 5-41
HFS:
  Access rights 5-8
  Creating a directory 5-14
  File names 3-18
  File permissions 15-40
  HFSCK utility 21-10
  Installing driver modules 18-23
  Introduction 3-25
  MKHFS utility 21-2
  OSINSTALL utility 21-5
  Overview 21-1
  Setup 21-1
  System volume 18-68
HFSCK utility (HFS) 21-10
Hierarchical directories (HFS) 3-25
Hierarchical directories (SRM) 3-20
High-speed disc interface (configuration) 18-18
Highlight Characters C-12, C-13
History of the system B-1

i

ID PROM 22-7
IF (Compiler option) 6-34
IMPLEMENT 8-30
IMPORT 8-30
IMPORT text 7-6
IMPORT text command (Librarian) 8-18
INCLUDE (Compiler option) 6-35
INCLUDE files (Compiler) 6-18
Incremental backup 20-4
Initialization Library file 18-5
Initialize Command (Main Level) 2-6
Initializing discs 15-42, 18-27
Initializing modules 7-13
INITLIB file:
  Adding modules for SRM 18-80
  adding modules to 18-38
  Introduction 18-5
  Library 18-7
  module descriptions 18-38
  renaming 18-35
  required order of modules 18-38
Input formats (Debugger) 9-12
Insert command (Editor) 4-37
Integer number range 13-1
Interchange of files B-18
Interface text 7-5, 7-6
Interface:
  Drivers 18-41
  HP 98265 SCSI bus interface card 18-3, 18-18
  HP 98546 Display 22-10
  HP 98620 DMA 18-16
  HP 98622 GPIO 18-16
  HP 98624 HP-IB 18-16
  HP 98625 (configuration) 18-3
  HP 98625 High-speed disc (HP-IB) 18-16, 18-18
  HP 98626 RS-232 serial 18-16
  HP 98627 Color output 18-16
  HP 98628 Datacomm 18-16
  HP 98629 SRM 18-17
  HP 98630 Breadboard 18-17
  HP 98635 Floating-point math 18-17
  HP 98643 and built-in LAN 18-17
  HP 98644 RS232 serial 18-17
  HP 98646 VMEbus 18-17
  HP 98658 SCSI bus interface card 18-3, 18-18
  HP built-in Parallel interface 18-17
  HP-Human Interface Link (HP-HIL) 18-44
Interleave, discs 15-41

I/O:
Addresses ... B-20
Library errors ... A-6
Memory map ... B-20
System errors ... A-4
IOCHECK (Compiler option) ... 6-36
IORESULT function ... 17-2, 17-3

j

JSR instruction (68000) ... 6-56
Jump command (Editor) ... 4-40

k

Katakana display characters ... C-10
Kernel (operating system) ... 18-5
Key notations (Debugger) ... 9-5, 9-30
Key notations in text ... 2-2
Keyboards ... 22-6
Knob ... 4-8, 4-26
Krunch command (Filer) ... 5-44

l

Language extensions ... 10-10
Length of strings ... 14-1
LIBRARIAN file ... 8-31
Librarian:
Adding modules to System Library ... 8-10
Command summary ... D-6
Creating a boot file ... 8-23
Creating libraries ... 8-12
DEF table ... 8-27
DEF table command ... 8-18
Define Source ... 8-28
Detailed file information ... 8-17
EXPORT ... 8-29
EXT table ... 8-28
EXT table command ... 8-18
Flags ... 8-29
General Value or Address Record (GVR) ... 8-30
Glossary ... 8-27
IMPLEMENT ... 8-30
IMPORT ... 8-30
IMPORT text command ... 8-18
Introduction ... 8-1
Invoking ... 8-10
Libraries ... 8-2
Linking object files together ... 8-14
Mass storage requirements ... 18-45

Mass storage setup 8-9
Object file 8-31
Object module 8-31
REF tables 8-32
Reference Pointer 8-32
Text Record 8-33
Unassemble commands 8-19, 8-21
What it does 8-3
Libraries 18-7
LIBRARY file 8-1, 8-31
Library:
Definition of 8-31
Overview 8-2
System 8-33
LIF file names 3-10
Line length limitation 4-51, 15-9
LINENUM (Compiler option) 6-37
LINES (Compiler option) 6-38
LINK instruction (68000) 6-56, 7-11
Linking object files 8-14
LIST (Compiler option) 6-39
List-directory command (Filer) 5-6, 5-46
Listing of files 5-11
Loader/Segmenter errors A-8
Loading a system 18-4
Local printers (CTABLE) 18-53
Local variables 7-11
LOCKABLE files 15-34
Logical unit numbers 18-8
Logical units 3-3
Logical volumes (hard discs) 18-11, 18-14

m

Main Command Level 2-1, 2-4
Main Command Prompt 2-1
Main Level Command summary D-1
Main Level commands 2-3, 2-4
Make command (Filer) 5-48
Manual overview 1, 4
Margin command (Editor) 4-41
MARK and RELEASE 16-2
Mass storage:
Comparison of 19-2
Configuration 18-17
Introduction 3-1
Volumes 3-2
MAXPOS (files) 15-30
MEDIAINIT program 15-42, 18-27
Memory map:
RAM B-19

ROM B-20
Software B-22
Memory volume command (Main Level) 2-7
Memory volumes 18-45
Memory-mapped I/O B-20
Memory:
Bubble 18-19
Characteristics 3-1
EPROM 18-19
RAM 3-1
Mixing DISPOSE and RELEASE 16-4
MKHFS utility 21-2
Modules:
Assembler 7-5
Developing and testing 6-10
Device drivers 18-5
Examples of 6-9, 6-10, 7-41, 7-42, 8-3
How the Compiler finds them 8-5
How the loader finds them 8-6
Importing 8-5, 8-6
Initialization 7-13
INITLIB 18-5
INITLIB module descriptions 18-41
LAST (in INITLIB) 18-6
Names used by operating system B-18
Object (definition of) 8-31
Pascal 8-32
PRINTER 18-17
Required order in INITLIB 18-39
Separate compilation 6-13
Strategy for compiling 6-14
Structure of 6-7
Mouse input device 4-8, 4-26, 18-44
Moving files 5-16
Multiple on-line systems 18-15

n

New command (Filer) 5-50
New sysvol Command (Main Level) 2-9
Non-disc mass storage (introduction) 19-1

o

Object file 8-31
Object module 8-2, 8-31
On-Line devices 2-6
Opcodes (Assembler) 7-19
OPEN (files) 15-5, 15-24, 15-25, 15-34
Operating system kernel 18-5
OPTIONS module (CTABLE) 18-52

OSINSTALL utility (HFS) 21-5
Other manuals 1, 2
Overview 1
OVFLCHECK (Compiler option) 6-40

p

Page command (Editor) 4-42
PAGE (Compiler option) 6-41
PAGEWIDTH (Compiler option) 6-42
Parallel printers 18-21
Parameter passing 6-58
PARTIAL_EVAL (Compiler option) 6-43
Partitioning hard discs:
  Algorithm 18-11
  Designing your own 18-59
  Example 18-27
  Modifying 18-14, 18-55
  Recommendations 18-58
Pascal:
  1.0 (description) B-1
  2.0 (description) B-2
  2.1 (description) B-2
  3.0 (description) B-5
  3.01 (description) B-9
  3.1 (description) B-10
  3.12 (description) B-14
  3.2 (description) B-14
  3.22 (description) B-17
  Extensions 10-10
  File operations 15-23
  Modules 8-32
  Program development 6-2
  System history B-1
  Volumes 3-2
Passing parameters 7-8
Passwords (SRM) 3-8
Peripheral drivers 18-41
Peripherals supported (by 3.0 system) B-7
Permanent Command (Main Level) 2-10
Permissions (HFS) 15-40
Physical memory map B-19
Porting software 22-1
POSITION (files) 15-30
Prefix command (Filer) 3-5, 5-51
Prefix volume 3-4
Primary DAMs 18-21
Printers:
  Changing the System Printer 18-18
  Modules 18-17
  Serial devices 18-19

Parallel 18-21
Problems:
Compiler 6-19
Debugging programs that use files 15-40
File names to avoid 3-9
Insufficient global space 6-20
No room on volume 3-19
Not Enough Memory 6-20
Syntax errors 6-4
TABLE program 18-13
Procedure calls (effects on stack) 6-56
Procedures (EXTERNAL) 7-16
Processor boards 22-3
Program development 6-2
Programming system 7-5
Prompts:
Compiler 6-3
Date 2-17
Debugger 9-7
Editor 4-3
Filer 5-2
Librarian 8-8
Main Level 2-1
Time 2-17
PUT (files) 15-27

q

Queue (Debugger) 9-9
Quit command:
Editor 4-43
Filer 5-53
Librarian 8-8

r

RAM:
Addresses B-19
Introduction 3-1
Memory map B-19
Random access files 15-28
RANGE (Compiler option) 6-44
Range of addresses B-20
Range of numbers 13-1
READ (files) 15-26
READDIR (files) 15-28
Real number range 13-1
REF (Compiler option) 6-45
REF tables 8-32
Reference Pointer 8-32
Relaxed typechecking (Compiler) 12-2

RELEASE and MARK 16-2
Remove command (Filer) 5-54
Removing files 5-19
Renaming:
  Boot files 18-35
  Files 5-18
  Volumes 5-18
Replace command (Editor) 4-45
RESET (files) 15-5, 15-25, 15-34
Restore (BACKUP) 20-5
REWRITE (files) 15-5, 15-24, 15-26, 15-34
ROM memory map B-20
ROM:
  Boot ROM 18-4
Run Command (Main Level) 2-11
Run-time errors A-3

S

Save Command (Filer) 5-56
SAVE_CONST (Compiler option) 6-46
SCANSTUFF module (CTABLE) 18-63
Screen dumps (Debugger) 9-8
SCSI bus driver (SCSIDVR) 18-19
SCSI disc considerations 18-19, 18-63
SCSI disc driver (SCSIDISC) 18-19
SCSIDISC 18-19
SCSIDVR 18-19
SCSI interfaces scanned 18-66
SCSIscanstuff module (CTABLE) 18-65
SCSI select codes searched 18-63
SEARCH (Compiler option) 6-47
SEARCH_SIZE (Compiler option) 6-48
Secondary DAMs 18-21
SEEK (files) 15-29
Self test (during boot) 18-4
Serial printers 18-19
Set command (Editor) 4-48
Single stepping a program 9-7
Slow program execution 9-7
Software memory map B-22
Software porting 22-1
Source program 6-2
Special configurations:
  Definition of 18-1
  Example changes 18-2
  Examples of 18-14
SRM:
  Access rights 3-8, 5-8, 15-39
  Concurrent file access 3-24, 15-37
  Configuration requirements 18-21

Creating a directory 5-14
Current working volume 3-22
Default volume 3-22
Directory configuration 18-75
Directory structure 3-20
Example configuration 18-72
File names 3-18
File notation 3-21
Hardware setup 18-73
Installing driver modules 18-74
LOCKABLE files 15-34
Multi-disc 18-83
Multiple unit numbers 18-65
Overview of installation 18-73
Passwords 3-8
Unit numbers 3-22
Volumes 3-22
Stack frame (Debugger) 9-17
Stack (How Pascal uses it) 6-55
STACKCHECK (Compiler option) 6-49
STACKCHECK_ON (Compiler option) 6-56
Standard configurations (definition of) 18-1
Standard partitioning, hard discs 18-57
STARTUP file 18-6
STARTUP file, renaming 18-35
Static and dynamic links (Debugger) 9-23
Static links 6-59
Stream Command (Main Level) 2-12
Stream files 4-24, 18-7, 18-37
String length 14-1
Strings and textfiles 15-34
Subsystems 2-1
Suffix Suppression 3-15
Suffixes 3-13
Summary:
Debugger commands 9-34
Editor commands 4-25
Filer Commands 5-21
Librarian commands 8-24
SWITCH_STRPOS (Compiler option) 6-50
Symbols (Assembler) 7-22
Syntax diagrams (introduction) 2-4
SYSPROG (Compiler option) 6-51, 17-1
System addresses B-22
System Boot file (SYSTEM_P) 18-5
System Boot files (naming) 18-15
System Boot files (renaming) 18-35
System discs 18-5
System files:
Copying 18-34

New sysvol command 2-9
What command 2-18
System history B-1
System Library 2-5, 2-9, 2-10, 2-18, 8-1, 8-33
System programming (extensions) 17-1
System version 2-16
System volume:
Bubble cards as 19-8
Definition 3-4
EPROMs as 19-20
New sysvol command 2-9
Search algorithm 18-61
SRM 18-80
TABLE selection 18-13
Volume listing 5-3
What command 2-18
SYSTEM_P file 18-5, 18-35
Systm files 18-4

t

TABLE program:
Auto-configuration 18-7
CTABLE source file 18-51
Failures 18-13
Initialize command 2-6
Modifying (CTABLE) 18-50
Renaming 18-35
Table-of-contents (BACKUP) 20-7
TABLES (Compiler option) 6-52
Tape backup utility 20-1, 20-9
Tape drives:
Access methods 19-35
Backup utility 20-1
Certify 20-13
File System access 20-13
Introduction 19-35
List of supported devices 19-35
Media-copy 20-11
Selective backup 20-15
Terminology 20-9
Verify 20-12
Volume backup 20-15
Technical Reference B-1
Text files:
Creating 4-3, 4-4
Declaration 15-30
I/O 15-30
Representation 15-31
Strings 15-34
Structure of 4-23

Text Record 8-33
Translate command (Filer) 3-15, 5-11, 5-57
TRAP instruction (68000) 6-56, 7-11
TRY/RECOVER 17-1

u

UCSD (Compiler option) 6-53
Unassemble commands (Librarian) 8-19, 8-21
Unblocked devices 18-10
Unit directory command (Filer) 5-60
Unit numbers:
  How assigned 18-9
  Initialize command 2-6
  Logical units 3-3
  Memory volume command 2-7
  New sysvol command 2-9
  Standard assignments 18-9
  Unit table 18-8
Unit Table 2-6, 2-9, 3-7, 18-8
Units, Blocked and Unblocked 3-3
UNLK instruction (68000) 6-57
U.S. ASCII characters C-2
User restart Command (Main Level) 2-15
U.S./European display characters C-4
Using the stack 7-15

v

Variables, (size of) 11-5
Variables:
  Global 2-10, 2-15, 2-17
  Zeroing 2-10, 2-15
Verification (BACKUP) 20-10
Verify command (Editor) 4-52
Version Command (Main Level) 2-16
Video compatibility hardware 22-10
Volume ID 3-7
Volumes command (Filer) 3-2, 5-3, 5-61
Volumes:
  Auto-configuration 18-8
  Backing up 5-13
  Default volume 3-4
  General 3-2
  Pascal 3-2
  Prefix volume 3-4
  PRINTER 18-18
  Renaming 5-18
  Specification (syntax) 5-25
  Syntax of identifier 3-7
  System volume 3-4

W

WARN (Compiler option) 6-54
What command (Filer) 5-62
What command (Main Level) 2-18
Wildcards 3-16, 5-10
Workfile 4-24, 5-20, 6-6
WRITE (files) 15-27
WRITEDIR (files) 15-29
WS1.0 file names 3-18

Z

Zap command (Editor) 4-55
Zero command (Filer) 5-63