HEWLETT
PACKARD

HP 9000
Series 200 and 300
Computers

Pascal 3.2 Workstation System Volume 1: Main Command Level and Subsystems

Pascal 3.2 Workstation System Manual

Vol 1: Main Command Level and Subsystems

HP 9000 Series 200 and 300 Computers

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

January 1987   **Edition 1.**

September 1987   **Update.** Updated to include caution against changing HFS default parameters with version 3.2.

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

March 1989   **Edition 2.** This edition 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.

December 1991   **Edition 4.** This edition includes information for the 3.24 and 3.25 releases of the Pascal Workstation.

Table of Contents

Chapter 4: The Editor

Chapter 6: Pascal Compiler

Chapter 7: The Assembler

Manual Overview 1

Introduction

This is Volume I of a two-volume manual that describes using the Pascal 3.2 Workstation System. It shows how to use the subsystems of the Pascal Workstation "environment" — the Editor, Filer, Compiler, Assembler, Librarian, and Debugger — and how they interact to provide you with a powerful Pascal program development tool.

Volume II focuses on programming and configuration of the system.

Before Reading this Manual

Here are the manuals that you should have read *before* reading this manual.

Documentation Guide

This guide describes each manual in the documentation set. It will help you to learn where the various parts of the system are described.

Computer Installation Guides

You should have already set up your computer hardware according to the instructions in the *Installation Guide* for your particular computer.

Peripheral Installation Guide

If you have peripheral devices such as disc drives and printers, you should have set them up according to the instructions given in this manual. It contains pertinent information taken from each supported peripheral's installation/operating manual.

Pascal User's Guide

You should have booted the Pascal system according to the instructions in the *Pascal User's Guide*. This manual also describes the software configuration required for various peripheral devices.

You may have also followed along with the examples to learn how to begin using the system to compile a few simple Pascal programs, although that is not mandatory due to similar coverage in this manual.

Pascal Textbook

If you are not familiar with the Pascal language, you should read *An Introduction to Programming and Problem Solving with Pascal* (included in the manual set sent with your system).

Volume II: Programming and Configuration Topics

The second volume of this manual is similar to the Pascal textbook described in the last paragraph, but it presents programming techniques that are specific to the Workstation Pascal programming language; i.e., the extensions to "standard" Pascal that are provided by the Workstation. As with the Pascal textbook mentioned above, you may want to read or scan Volume II of this manual before delving too deeply into this volume.

Previous Workstation Pascal Manuals

If you are familiar with the documentation for earlier versions of the Pascal Workstation, you may be happy to know that this manual is a later edition of the original *Pascal User's Manual*. However, this manual describes only version 3.2 of the Pascal Workstation.

Note

The main text of this manual does not generally discuss earlier versions of the system; however, the "System History" section of the "Technical Reference" appendix of this manual will help you if you are upgrading from an earlier version of the Workstation Pascal System.

Other Manuals

This manual does not generally assume that you are familiar with any of the other languages and systems available for this series of HP computers, although references are occasionally made to some of these other languages where appropriate (such as BASIC or HP-UX).

Previews of Chapters in Volume I

Here are brief previews of the contents of each of the chapters of this manual.

Chapter 2: The Main Command Level

This chapter describes the commands available in the "Main Level" of the Workstation System.

Chapter 3: The File System

This chapter introduces you to the Workstation File System. It describes how the logical units and volumes are organized, and gives a description of the various file system types which are available. However, it does not describe access of files from Pascal programs, as this is covered in the "Programming with Files" chapter in Volume II.

Chapter 4: The Editor

A program usually starts out as an idea. The Editor's function is to provide a useful environment for the translation of thoughts into actual programs or documents. This chapter fully explains the features of the Pascal Workstation Editor.

Chapter 5: The Filer

The Filer is used to store, load, copy, translate and perform other file-related utility operations. This chapter details performing these operations with the Filer.

Chapter 6: The Pascal Compiler

Once a program has been written with the Editor, this source code must be compiled into object code before it can be executed. This chapter explains the operation of the Compiler and the options that can be used to modify its operation. The chapter also describes the modular programming capability, which is one of the most powerful features of this system.

Chapter 7: The Assembler
This chapter introduces you to the Assembler, which converts programs written in assembly language — a humanly understandable version of the microprocessor's machine language — into object code for the MC68000 family of processors used in these HP Computers.

Chapter 8: The Librarian
This chapter covers the use of the Librarian. In the Pascal Workstation are libraries of object-code modules: some consist of device-drivers, while others consist of useful procedures for such applications as I/O and graphics. You can also design your own modules. The Librarian's function is to manage libraries of Pascal and Assembler language object modules.

Chapter 9: The Debugger
We all wish that a program would run perfectly the first time. Unfortunately, there is little evidence in real life to support that fantasy. The next best thing is to have some good tools to help you debug your programs. This chapter explains the debugging features available with this system.

Error Messages
This appendix contains the complete listings of all error messages for the various Pascal sub-systems.

Index
This section contains an index to the topics in both volumes of this manual.

Previews of Chapters in Volume II

Here are brief previews of the contents of each chapter in the second volume of this book.

Chapter 10: Overview of Workstation Software Features
This chapter gives a brief introduction to the language and library features of the Workstation. It also tells where various software features are described in the Workstation Pascal documentation set.

Chapter 11: Data Types and Structures
This chapter describes the types of data available in the Workstation Pascal language. It also briefly describes some of the data types and structures that might not be available with other implementations of Pascal.

Chapter 12: Program Flow
This chapter describes the features of Pascal language which allow you to alter the flow of a program.

Chapter 13: Numeric Computation
This chapter describes the standard Pascal numeric data types and how they are implemented on the Workstation. It then shows several examples of useful techniques for dealing with angles, rounding, logarithms, number-base conversion, calendars, and pseudo-random numbers.

Chapter 14: String Manipulation
This chapter describes how to use the HP Pascal type **string**, as well as using the associated string functions and procedures.

Chapter 15: Programming with Files
This chapter describes general uses of files, as well as many Workstation-specific file access techniques.

Chapter 16: Dynamic Variables and Heap Management
This chapter describes how to create and use dynamic variables, as well as how to reclaim the memory used for these temporary variables.

Chapter 17: Error Trapping and Simulation
This chapter describes how to programmatically handle, and possibly correct, errors before they halt the execution of your programs. It also shows how to simulate errors in order to debug the error-handling portions of your programs.

Chapter 18: Special Configurations
This chapter describes how to set up "non-standard" configurations. It first gives background information regarding how the system boots and configures itself, and then it describes the steps required to set up several configurations.

Chapter 19: Non-Disc Mass Storage
Several "non-disc" types of mass storage devices are available on the Pascal Workstation: EPROM (Erasable Programmable Read-Only Memory) cards, Magnetic Bubble Memory cards, and cartridge tape drives. Configuring and using these devices is described in this chapter.

Chapter 20: Backup Utilities
This chapter describes the methods and the utilities supplied which enable you to make a backup of all or selected data on a supported disc.

Chapter 21: HFS Setup and Utilities
This chapter describes how to use the utilities supplied with the system which enable you to create and check an HFS on your disc, and enable you to boot from an HFS disc. HFS is short for Hierarchical File System, the file system used by Series 300 HP-UX (5.0 and later versions).

Chapter 22: Porting to Series 300
This chapter describes the approaches available for running existing Pascal 3.0 software on Series 300 computers with the Pascal 3.2 system.

Technical Reference Appendix
This appendix contains the following information:

- A history of the Pascal system, which includes descriptions of the differences between the 3.2 version and previous versions of the Workstation system

- A list of module names used by the 3.2 system

- Software memory map

- Tables of available display characters.

Command Summaries
This appendix contains a summary of commands for each of the Pascal subsystems.

Glossary
Knowing what technical terms mean is always useful.

Error Messages
This appendix is an abbreviation of the lengthier appendix given in Volume I. This listing of errors fits on a single sheet of paper, which you may find handy to remove from the manual and place in a more convenient place.

Index
This volume also has an index to the topics in both volumes of this manual.

The Main Command Level 2

Introduction

The Main Command Level is the central point of reference for the operating system. It is "where you are" after booting the system and entering the time and date.

All the Main Command Level commands are listed in the subsequent Quick Reference. However, this chapter focuses mainly on those Main Command Level operations which do *not* call subsystems (such as the Editor, Filer, Compiler, etc.); each subsystem is described in later chapters of this manual.

Main Command Prompt

The Main Command Level consists of two prompt lines, only one of which is displayed at one time. Press the **?** key to toggle between them.

```
Command: Compiler Filer Editor Initialize Librarian Run eXecute Version ?
```

```
Command: Assembler Debugger Memvol Newsysvol Permanent Stream User What ?
```

The uppercase letters in the prompt lines indicate which key to press to start the operation.

All of the operations are available regardless of which prompt is being displayed.

The prompts are abbreviated on the 50-column display of the Model 226.

```
Command: Cmplr Edit File Init Libr Run Xcut Ver ?
```

```
Command: Asm Dbg Memv New Perm Stream User What ?
```

Keyboards

The Pascal Workstation supports several different styles of keyboards on various models of HP 9000 Series 200 and Series 300 computers. Descriptions of each keyboard are presented in the *Pascal User's Guide*. Alphanumeric keys are the same on all keyboards, but some of the special keys have different labels on the different styles of keyboards.

When this manual indicates two special keys, the first is for the 4602x style of keyboards and the second for the 98203 style of keyboards. You may wish to consult the "Key Correspondence Table" from the first chapter of the *Pascal User's Guide* when using the 98203 style of keyboard.

When you are directed in this manual to press a special key, the text will usually say: "Press the [Return] or [ENTER] key."

Another common example is the [Select] key on the 4602x keyboards and the [EXECUTE] key on the 98203 keyboards. When you are directed to press one of these keys, the text will say: "Press the [Select] ([EXECUTE]) key." (The second key noted in parentheses is the 98203 key.)

If you have a 4602x keyboard, you will note that there are both a [Return] key **and** an [Enter] key. The Pascal Workstation treats these two keys the same.

Main Command Quick Reference

Command	Description
Compiler	Calls the Compiler to translate Pascal source code into object code.
Editor	Calls the Editor for creating or editing a source program or textual document.
Filer	Calls the Filer for management of the File System.
Initialize	Initializes the File System (but not discs).
Librarian	Calls the Librarian for managing, linking, or unassembling object-code files.
Run	Runs the workfile (compiling it if needed) or the last program compiled or assembled since power-up. If there is no workfile, Run operates like eXecute.
eXecute	Asks for a code file and runs it.
Version	Allows setting the date, time, and time zone, and displays all the current system version information.
Assembler	Calls the Assembler to translate an assembly language source program into object code.
Debugger	Runs a program under control of the Debugger.
Memory volume	Sets up a memory resident mass storage volume for fast access.
New sysvol	Asks for a volume to be designated as the system volume.
Permanent	Asks for a code file to be permanently loaded into memory for execution without disc loading each time.
Stream	Asks for a stream file whose characters are interpreted as keyboard input until there are no more left.
User restart	Restarts the last program or subsystem that was run.
What	Displays the system file table and allows you to change the system files or system and default volumes.

Main Command Reference

Each command in this section contains a description and a syntax diagram. The syntax diagrams contain rounded and rectangular boxes. Elements in rounded boxes should be interpreted as literals. An example is as follows:

This notation indicates that you must literally type a ⬛C as part of the command.
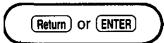
The ⬛Return or ⬛Enter indicates that you can press either key.

Elements in rectangular boxes are non-literal descriptions of command parameters. An example is as follows:

This notation indicates that you must supply the actual file name as part of the command.

An example of a complete command is as follows:

If, for example, this was the Compiler command syntax diagram, it would mean that you must type ⬛C to run the Compiler, then type the name of the file to be compiled, and enter the file name with either the ⬛Return or the ⬛ENTER key.

eXecute

The eXecute command runs a specified code file.
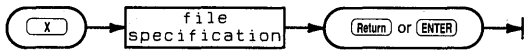
```
( X )──▶┌──── file ────┐──▶( (Return) or (ENTER) )──▶│
        │specification│
        └─────────────┘
```

Item	Description	Range
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

The file you specify should be previously compiled or assembled and ready to run. It is not necessary to include the .CODE suffix in the file name; it is automatically appended to the file name if not included. If the actual file name does not contain a .CODE suffix, you will need to terminate the file specification with a period to suppress this suffix.

If the specified code file imports other modules not found in the file, those modules must be contained in the current System Library (which must be on-line) or they must be Permanently loaded (by using the Permanent command at the Main Level). You can use the What command to see which file is designated as the current System Library, and to change it if desired.

Initialize

The Initialize operation updates Unit Table entries for all units that are currently on-line. (It does not initialize mass storage media; that function is performed by using the MEDIAINIT utility program. See the *Pascal User's Guide* for further details.)

Semantics

The Unit Table contains a record for each of 50 possible logical units available to the File System. The assignment of unit numbers to physical devices (auto-configuration) is performed by the TABLE program at power-up. Each record contains the "device address vector" of the physical device which corresponds to that logical unit number. The computer looks at the physical location indicated by the device address vector to see if the device is on-line. If it is, that fact is marked in the record for that unit, along with the volume name (if media is currently installed in the device). When you press ▭ the computer only looks at the Unit Table to see if a particular device is on-line; it does not check the actual device. (See the Booting Process section of the Special Configurations chapter for further details of how the TABLE program works.)

When a device is added to your system after the computer has been powered-up, you will usually need to execute BOOT:TABLE or power-up the system again in order for the device to be recognized. However, the Initialize command may in some cases be sufficient to get the system to recognize the new device.

Initialize also performs a device clear for all on-line devices and causes the system to forget the last loaded file (the User command can't reload the last program). The Initialize operation also causes all temporary files to be removed from each volume the next time a file is opened on the volume.

The volumes CONSOLE: (Unit #2) and PRINTER: (Unit #6) are special cases; these volumes are always assumed to be on-line. Thus, the system may "hang" if either of them is off-line when you try to access them.

Memory volume

The Memory volume command creates a mass storage volume in memory.
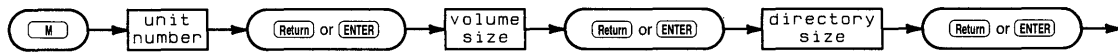
Item	Description	Range
unit number	integer	7 thru 50
volume size	integer indicating the number of 512-byte blocks	≥ 1
directory size	integer indicating the maximum number of files in the volume	≥ 1

Semantics

The Memvol command gives you the capability for very fast mass storage operations.

When the Memvol command is given, you are prompted for a unit number. This number corresponds to an entry in the Unit Table. Don't give a unit number which is already in use. The Volumes command in the Filer subsystem shows which unit numbers are currently used. For most applications, 50 is the recommended unit number to use for your first memory volume.

You are then prompted for the number of (512-byte) blocks needed for the memory volume. Try to estimate conservatively the amount of memory you want reserved for the memory volume because it cannot be returned for general purpose use without turning off the computer. On the other hand, if you don't specify enough space, you have to create another larger volume.

Memory volumes are useful for program development where a lot of mass storage I/O is involved (e.g. editing and compiling). Reserve enough space on the memory volume for both the source file, the object code file, and 40 extra blocks for the Compiler's temporary files. A good rule of thumb for LIF memory volumes is:

size_of_volume = size_of_source_file (in 512-byte blocks) * 4 + 40

If you are transferring a source file from disc (as opposed to starting from scratch) you can determine its size by getting a directory listing of the volume that contains it. Use the **bytes** or **blocks** value to determine the file size and add a "fudge factor" so the file can grow in size. Note that different file systems return the file size in different units.

Note that the default directory access method (DAM) for memory volumes is LIF; this DAM is the primary DAM specified in the TABLE program. See the Special Configurations chapter for further details about changing the primary DAM.

You are then prompted to give the number of directory entries you need for this memory volume.

`Number of directory entries ?`

Type the number you think you'll need and press `Return` or `Enter`.

You can refer to your memory volume by its unit number (e.g. `#50:`). Alternately, you can refer to it by its given volume name, which is initially RAM: (e.g. MYRAM:).

If you plan to use more than one memory volume, use the Filer's Change command to give each memory volume a unique name.

Here is a method for setting up an extremely fast program development environment.

1. Create a RAM: volume and specify it as the system volume using the Newsysvol command.

   Specify RAM: as the default volume using the Main Command Level's What command or the Filer's Prefix command.

2. Permanently load the Editor and Compiler using the Permanent command.

3. Go into the Editor and write your program.

4. When you're ready to leave the Editor, use the Update option to create a workfile. The system puts the workfile on the fast RAM system volume.

5. Press `R`.

Your file will automatically be compiled. If it compiles with no errors, it will be run. If it contains errors, you will have the option of returning to the Editor.

Note

Since memory volumes are volatile, don't forget to save the files in the memory volume on a disc before turning off the computer.

New sysvol

The New sysvol command specifies a new system volume and updates the operating system file table accordingly.

```
( N ) → [unit number] → ( (Return) or (ENTER) ) →
```
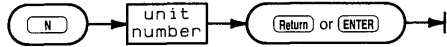
Item	Description	Range
unit number	integer	1 thru 50

Semantics

The system file table is used in locating operating system files. It contains the volume and file names of system files (EDITOR, FILER, etc.). When you press a key at the Main Command Level that invokes one of these subsystems (such as $\boxed{E}$), the system attempts to load the corresponding file indicated in the system file table (here, the EDITOR file).

You can use this command to specify a new system volume. The first step in this operation prompts you for a unit number. The device corresponding to the specified unit number is considered to be the new system volume, and serves as a starting point in the search for the system files: ASSEMBLER, COMPILER, EDITOR, FILER, LIBRARIAN, LIBRARY, and the work file. If any of these system files are not found, the Unit Table is used in a sequential search for the rest of them. As each file is found, the name of the volume on which it is found is prepended to the file name (for instance, SYSVOL:LIBRARY), and the complete file specification is placed in the file table. If any system file is not found in this search, the operating system assumes that it will find the file on the flexible disc volume on which it was delivered (for instance, ACCESS:EDITOR).

Use the Main Command Level's What command to see the resultant system file table.

If the system date is 1 JAN 70 at the time this command is given, the system date will be set using information stored on the volume.

If the time is hour zero and minute zero and the new system volume is an SRM, the system time will also be set.

Permanent

The Permanent command loads a program permanently into memory.

```
( P )→| file
       specification |→( Return or ENTER )→|
```

Item	Description	Range
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

The Permanent command can be used to load a user program, a system program (Editor, Compiler etc.), or a module that is needed by a program. This code file is then ready to execute immediately when the command is given. A "P-loaded" (Permanently loaded) program does not have to be loaded from disc each time it is run.

After you give the Permanent command, you are prompted for the name of the file which contains the module or program. You need not include the .CODE suffix; if you don't include one, the suffix will be appended to the file name. If the file to be P-loaded does not have a .CODE suffix, end the file specification with a period to suppress the suffix from being appended to the file name automatically.

Several programs may be P-loaded in memory. The operating system keeps track of which programs have been P-loaded. When you give a command to run a program, the operating system checks to see if it has been P-loaded; if so, it is executed immediately. If not, it is loaded from disc and then executed; in this case, the memory used by the program is reclaimed after execution terminates.

An object module which is imported by a program must either be P-loaded or contained in the current System Library (which must be on-line).

A program or module's global variables are zeroed only when it is loaded, not each time the program is run.

Note

The volume name is not retained when a file is P-loaded. Attempting to execute a file of the same name but on a different volume will still result in the P-loaded file being executed. If a pathname is included however, the P-loaded file is ignored and the file is searched for and loaded from the file system.

For SRM users and those with HFS discs, do not use a directory path name to execute a P-loaded file. See note above.

Run

The Run operation causes the workfile or last compiled program to be executed.
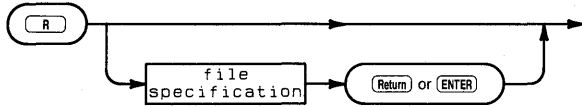
Item	Description	Range
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

When the Run command is given, the operating system checks to see if there is a workfile. If there is a CODE workfile, it is executed; if not, the most recently compiled or assembled file is executed. If there is a TEXT workfile but no CODE workfile, the TEXT workfile is first compiled (with the system compiler) to a CODE file and then the CODE file is executed. If there is no workfile or previously compiled program, the command operates like the eXecute command and you are prompted for a file specification.

Stream

The Stream command "executes" a file of ASCII characters as if they were being typed from the keyboard.
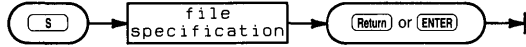
Item	Description	Range
file specification	literal	Any legal file specification (see the File System chapter)

Semantics

A command stream or stream file is a file that is interpreted as input to the Main Command Level and/or its subsystems in place of keyboard input. The Stream operation causes a file to be interpreted. Therefore, a stream file is useful for executing a sequence of commonly used commands without requiring any operator intervention.

A stream file is created with the Editor and may be of type TEXT, ASCII, HP-UX compatible or Data. If you do not specify a suffix, a ".TEXT" is automatically appended to the file name; if the name of the file to be streamed does not have a suffix, add a trailing period to the file name to suppress the suffix.

In order to generate a valid sequence of keystrokes, you should first run through the desired sequence while noting the keystrokes entered. Note particularly the occurrences or absences of the [Return] or [Enter] key. Then enter the same keystrokes in your stream file. If, during an Editor or Filer command sequence, you encounter an unpredictable question that has a (Y/N) or (R/O/N) question associated with it, do not answer the question in the stream file. These kinds of questions are answered automatically as the file is streamed. (Y/N) questions (Yes/No) are answered "Y". (R/O/N) questions (Remove/Overwrite/Neither) are automatically answered "R".

After all the characters in a stream file have been interpreted, control is returned to the keyboard.

Comments

Stream files may contain comments. A line beginning with an asterisk (*) will be interpreted as a comment if it occurs at the Main Command Level. (Comments cannot be embedded among commands for subsystems or user programs.) When the command interpreter encounters one or more comment lines while streaming, they are displayed briefly on the screen, thus allowing the process to be monitored.

Immediate Execute Keys

If it is necessary to use keys that also act as immediate-execute commands in the Editor, such as [Select] ([EXECUTE]) or [Back space], use the following key sequences to generate those keystrokes.

Immediate-Execute Key	Generate with these Keys
[Select]	[CTRL]-[Select] [C]
[Back space]	[CTRL]-[Select] [H]
[Tab]	[CTRL]-[Select] [I]
[Clear display]	[CTRL]-[Select] [L]
Left arrow	[CTRL]-[Select] [H]
Right arrow	[CTRL]-[Select] [<]
Up arrow	[CTRL]-[Select] [?]
Down arrow	[CTRL]-[Select] [J]

If you have a 98203 keyboard, substitute [EXECUTE] or [EXEC] for [Select] in the preceding table.

Prompts for Keyboard Input

A stream file can be made to display a prompt on the console and then wait for an input string from the keyboard. The input string is assigned to a variable in the stream file. When the variable is encountered during streaming, the string is used in its place.

This input prompting must appear in the stream file *before all* of the commands or comments. Up to 36 prompts are allowed. They are denoted with an "=" as the first character on a line.

To prompt for an input string, place an equal sign, followed by a single alphanumeric character variable name (uppercase and lowercase letters used for variables are treated as equal), followed by the prompt text. For example:

    =f What is the name of the file to be P-loaded ?

When the file name is typed in response to the prompt, it is stored in the specified variable, in this case the variable named f.

After the input prompting, begin entering the commands in the stream file. When you want the input string to be given to the operating system, use the variable preceded by "@". For example, the following characters are a command stream:

    p@f

The command is the Permanent load command with a file name parameter indicating which file is to be P-loaded. The file whose name was given in response to the above prompt is then P-loaded.

Stream Files on Read-Only Devices

Normally when a file is Streamed, the file is copied to a file named STREAM on the current system volume; during this copy, prompts are displayed and @ variables are assigned values input from the keyboard by the computer operator. After all variables have been assigned, the original file is closed and the STREAM file is read as keystrokes.

The stream file mechanism will not work if the current system volume is a read-only mass storage device or if the current system volume is completely full. To avoid creating the STREAM file on the system volume, you can add the three-character token [*] to the end of the file name you wish to stream. If, for example, you have a file named FIXIT.TEXT you wish to stream, your response to the system prompt Stream what file? would be:

    FIXIT[*]

By including the [*] token, the normal processing of the stream file will be turned off and no file will be created on the system volume. Note that this means your stream file cannot prompt for any input from the computer operator.

It is this mechanism (see preceding discussion) that allows the use of stream files stored on read-only mass storage, such as EPROM, and the use of read-only devices as system volumes.

This mechanism is also used to process the AUTOKEYS stream file, if found during the boot process when the AUTOSTART stream file is not present. For examples of AUTOSTART and AUTOKEYS stream files, see the discussions in the *Pascal User's Guide* and in the Special Configurations chapter of this manual.

User restart

The User restart command causes the last program that was run to be rerun.
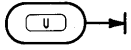
Semantics

Included in the meaning of "program" are user programs and operating system programs such as Editor, Filer, Compiler, etc.

Global variables are zeroed at the time a program is loaded, not each time a program is rerun.

Version

The Version operation allows you to change the system date, system time and system time zone.
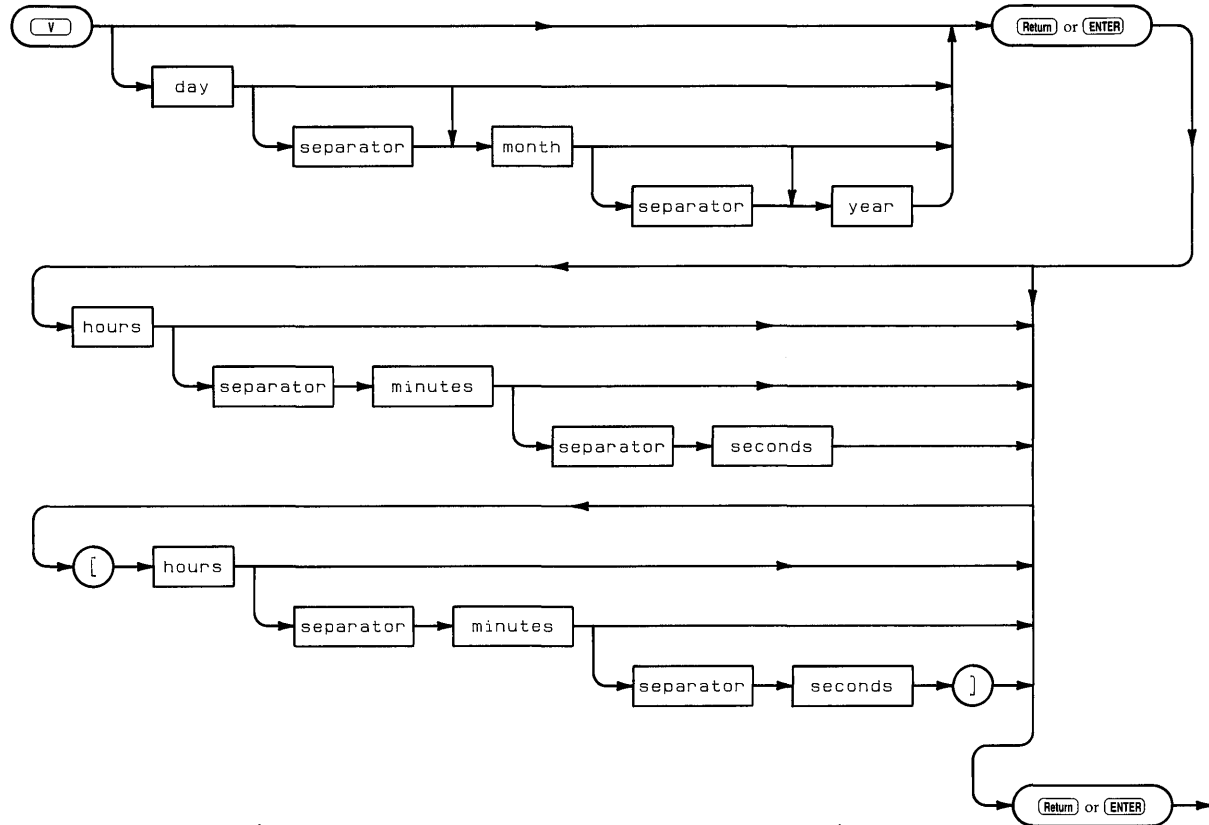
Item	Description	Range
day	integer	1 thru 31
month	three alpha characters; letter case is ignored	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
year	integer	0 thru 27, 2000 thru 2027 70 thru 99, 1970 thru 1999
hour	integer	0 thru 23
minutes	integer	0 thru 59
seconds	integer	0 thru 59
separator	non alphanumeric character	:, -, /, space, etc.

Semantics

In addition to prompting for the system time [and timezone] and date, some operating system information is displayed. The current operating system revision, available global and user memory space information, and default and system volume information is also displayed.

The system time is defined in local time, the timezone specifying the difference between Greenwich Mean Time (GMT) and local time. For example, a system using Central European Time (CET), normally one hour ahead of GMT, should have its Time Zone set to [-1:00:00].

This specification of the system time relative to GMT is most useful when transferring data and programs to and from HP-UX systems, as the HP-UX operating system runs under GMT. Unless you are absolutely certain you are not going to communicate with an HP-UX system in terms of transferring files, it is recommended that you set the timezone to the correct value.

The Version Prompt

```
New system date ?

System date is          21-Jan-87
Clock time is           14:14:50
Time Zone is            -1:00:00

Workstation             Rev.  3.2  15-Jan-87

Available Global Space 57960 bytes
Total Available Memory 191042 bytes

System  volume:  SYSVOL:
Default volume:  SYSVOL:

Copyright Hewlett-Packard 1982, 1983, 1984, 1985, 1987
Copyright A.T.&T. 1980, 1984
Copyright Univ. of California 1979, 1980, 1983
        RESTRICTED RIGHTS LEGEND
Use, duplication or disclosure by the U.S.
Government is subject to restrictions as set
forth in subdivision (b)(3)(ii) of the Rights in
Technical Data and Computer Software clause at
52.227-7013. Hewlett-Packard Company,
3000 Hanover Street, Palo Alto, CA 94304
```

For more details on "Global Space", see the Compiler chapter.

Here are some typical hour values for setting the time zone. Values shown are for standard time (subtract one hour for daylight savings time).

Time Zone	City	Value
	Honolulu	10
Pacific	San Francisco	8
Mountain	Denver	7
Central	Chicago	6
Eastern	New York	5
	London	0
	Paris	-1
	Cairo	-2
	Hongkong	-8
	Tokyo	-9
	Sydney	-10

What

The What command displays the "system file table" and allows you to specify new file specifications for the system files.
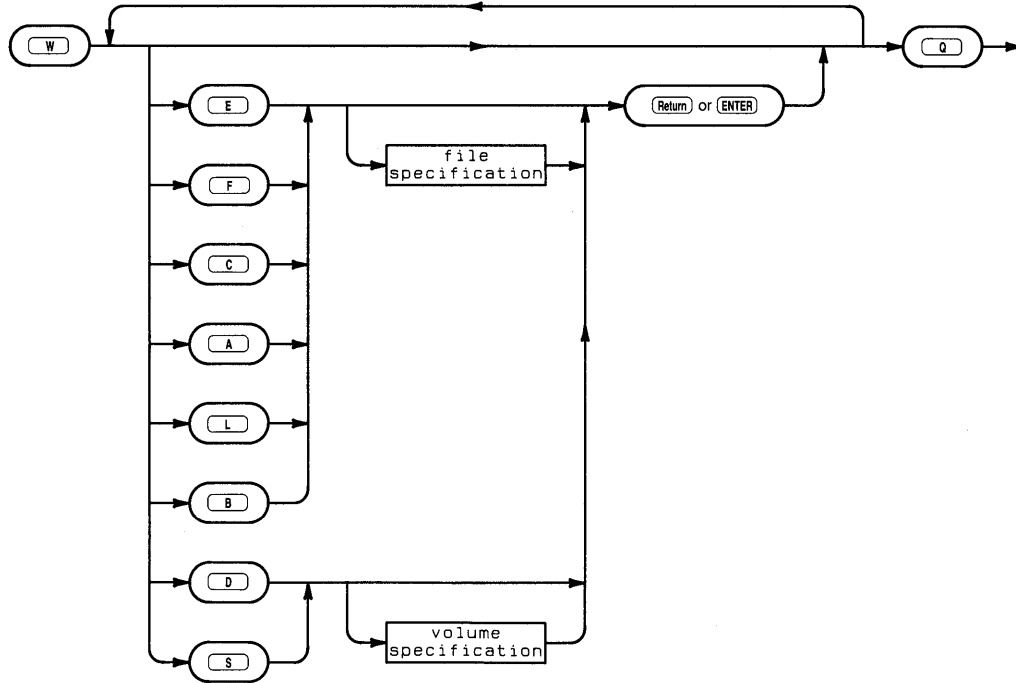
Item	Description	Range
file specification	literal	any legal file specification (see the File System chapter)
volume specification	literal	any legal volume specification (see the File System chapter)

Semantics

The system file table contains file specifications that are used by the operating system when locating system files (Assembler, Compiler, Editor, Filer, Librarian, Library, and Default and System volumes). The What command displays the system file table; a typical example is shown below.

```
Assembler Compiler Editor Filer Librarian
liBrary System volume Default volume Quit

ASSEMBLER     SYSVOL:ASSEMBLER
COMPILER      SYSVOL:COMPILER
EDITOR        SYSVOL:EDITOR
FILER         SYSVOL:FILER
LIBRARIAN     SYSVOL:LIBRARIAN
LIBRARY       SYSVOL:LIBRARY

* System  volume:  SYSVOL:
: Default volume:  SYSVOL:
```

Typing one of the uppercase letters at the top of the menu allows you to change the corresponding file specification for that system file.

Note

When specifying a system file name that does not have a .CODE suffix, use a period at the end of the file name to prevent a .CODE suffix from being appended to the file name.

With this command, it is possible to do such things as specify a file other than LIBRARY as the System Library, or your custom graphics editor as the System Editor. In the case of your custom editor, you need only press ⌈E⌉ to invoke it.

Specifying a logical unit number such as #3: as the Default volume allows *any* disc media in a unit with removable media to be the desired volume. To subsequently specify any volume in the default unit, only the file name need be specified. To accomplish this, make sure that the drive door is open (or the drive is empty), type ⌈D⌉ for a Default volume change, and then type the following:

#3: ⌈Return⌉ or ⌈Enter⌉

Note

If you are specifying a system volume, please refer to the New sysvol command which is listed earlier in this reference.

The File System

3

Introduction

This chapter introduces you to the Pascal Workstation File System. The File System organizes and accesses information which is stored on mass storage devices. Even if you are an experienced programmer, you should read this material because it will help you understand the features of your Pascal Workstation.

Primary vs. 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 external mass storage, also called **secondary storage**. Normally, data processed by the computer must first be placed in internal memory. (The term "data" is used broadly to mean any information processed by the computer, so programs are data, too.)

RAM has three important characteristics:

- RAM is very fast: Some data items can be stored or retrieved from RAM in less than a millionth of a second.

- RAM is volatile: Data in RAM is lost when the computer is powered off.

- It is expensive compared to alternative, slower forms of data storage, such as discs or magnetic tape.

Information not immediately needed by the computer is kept in secondary storage. Some important characteristics of magnetic discs are listed below:

- Data access is slow compared to RAM, often as much as ten thousand times slower.

- The data is relatively permanent; that is, it is available until erased.

- Magnetic storage is inexpensive compared to RAM.

- Magnetic media are often removable and replaceable, providing an almost unlimited amount of long-term storage.

Pascal Volumes

Let's take an exploratory trip, using the computer itself to investigate the file system. You should already know how to load the Pascal Language system, and be aware of its various subsystems, such as the Editor and Filer.

To begin our journey, begin at the Main Command level and load the Filer subsystem by pressing:

$\boxed{\text{F}}$

The Filer is located on the Pascal disc labelled **ACCESS:**. When the Filer's prompt line appears, execute the Volumes command by pressing:

$\boxed{\text{V}}$

The various disc drives connected to your computer will be accessed, and then you will see a display similar to the following:

```
Volumes on-line:
   1    CONSOLE:     This is your workstation's CRT display.
   2    SYSTERM:     This is your workstation's keyboard.
   3  # BOOT:        BOOT: disc is in right-hand drive (of a 236).
   4  # WRKING:      Initialized disc with volume name WRKING:
   6    PRINTER:     Printer is connected to built-in HP-IB.
  11  * SYSVOL:      The * indicates the system volume.
  12  # MYVOL:       (Volumes 11 through 14 are examples of a
  13  # MASTER:      possible HP 9133A disc drive configuration.)
  14  # V14:
  50  # RAM:         Ram volume made with Memory Volume command.
Prefix is - MYVOL:   MYVOL: is current prefix volume.
```

Precisely what will be displayed depends on the drives connected to your computer and what discs are currently installed in those drives. Note how your display appears. You may want to change some of the discs in your system's disc drives, or turn off a peripheral and see how that changes the display.

Volumes

The word "volume" was chosen by analogy to a book. Volume denotes a logical entity in which a substantial amount of information can be stored. For instance, a flexible disc is a volume. Volumes have names by which we may refer to them. The display above shows what volumes are currently accessible to the File System.

Notice that each volume name is followed by a colon. This convention is used throughout the Pascal system. The colon is a delimiter or punctuation mark which separates the volume name from further information used to designate data within the volume.

A single large disc may contain more than one volume, as a shelf can hold more than one book. Flexible discs usually contain a single volume. Thus, for flexible discs you may use the volume name as the disc drive's name. For instance, if we refer to the volume **BOOT:** by name, then the computer will find it in whichever drive it is located.

NOTE

Because the file system works with named volumes, it is very important not to have more than one volume of a given name on-line at one time. The File System may destroy data by using one volume when you meant the other.

You can see that some of the volume names don't correspond to a disc device. For instance, SYSTERM: is the name of the keyboard volume, CONSOLE: is the name of the CRT display, and PRINTER: is the name of the system hard-copy device. Actually, the File System has a name for each input/output device that it is able to access. We will have more to say about these non-disc volumes later.

Logical Units

The numbers in the column to the left of the volume names displayed above are called "logical unit numbers" or simply "units". The volume name denotes a particular disc, while the unit number denotes a particular location for a volume. In the case of flexible discs, the unit number corresponds to a physical disc drive. In the case of a large fixed disc which is divided into several volumes, each logical unit represents a portion of the disc surface which is treated as if it were a separate physical disc drive.

To refer to a unit instead of a volume, use a # followed by the unit number. For instance, on a Model 226 or Model 236 computer, #3: and BOOT: both refer to the same volume as long as the volume named BOOT: is installed in the right-hand disc drive.

Drive Numbers vs. Unit Numbers

Since a single machine can contain two or more drives, you need to be able to distinguish between them. If you read the machine's manual, you will find that the drives are differentiated by *drive number*. For instance, the right-hand floppy drive in a Model 236 is drive number 0, while the left-hand drive is drive number 1. The File System distinguishes between them by assigning each a unique *logical unit number*. In the case of the Model 236, these drives are normally assigned unit numbers 3 and 4, respectively. With external dual floppy drives, drive 0 is usually the left-hand drive, while drive 1 is the right-hand drive. And with hard disc drives, there can be several drive numbers. Don't be alarmed, however, because the system takes care of the correspondence between drive numbers and unit numbers for you. In addition, this manual refers almost solely to logical unit numbers, not drive numbers. Drive numbers were mentioned so that you would realize that they are not the same as unit numbers.

Blocked and Unblocked Units

Some of the units are displayed with # or * between the unit number and the volume name. These are blocked units. Blocked units are memory devices that are divided into sectors (blocks) and have directories describing their contents.

We aren't yet ready to talk about the data stored in a volume, but you probably won't be surprised to learn that it is organized into groups called "files", which are like chapters in a book. A directory on a blocked volume is essentially a table of contents.

The other units are unblocked or "byte stream" devices (such as the printer, keyboard and CRT). Unblocked devices process information one character at a time and do not have directories.

The System Volume and Default Volume

Although your workstation can deal with many volumes (up to 50 on-line at once), there are two volumes which are referred to so frequently that special abbreviations have been provided to name them. They are the *system volume* and the *default volume*.

The System Volume

The system volume is used by the Operating System to store its own private files and records. Since the Operating System is always overseeing your computer's operation, the system volume needs to be accessible practically all of the time. The abbreviated name for the system volume is * (asterisk), which appears next to the system volume in the Volumes command's display. The asterisk need not be followed by a colon, since it is distinctive. Thus for the Volumes display shown previously, these notations all denote the system volume:

```
*
*:
SYSVOL:
#11:
```

Here are some of the ways the operating system uses the system volume:

- When the Operating System is loaded and begins to function, it first looks on the system volume for subsystem programs such as the Editor, Filer and Compiler. However, if these subsystem programs are on other accessible volumes, the Operating System will still find them.

- When the Operating System first begins to function, it may look on the system volume to find the system date. The system date is put on all files as they are created to help in maintaining file organization. If you change the system date, the new date gets written on the system volume.

- During processing of a stream file, data is temporarily stored on the system volume. A stream file is a pre-recorded sequence of keystrokes which are treated as if they came directly from the keyboard.

- If you create an anonymous file (see the Programming with Files section), it will be stored on the system volume. An anonymous file is a file created by a program, used by the program, and then destroyed when the program ends. While the program is in existence, the anonymous file is for all purposes a real file.

- If you use a work file during development of a program, it will be stored on the system volume.

- If you use an AUTOSTART or AUTOKEYS file, it must be stored on the system volume.

The Default (Prefix) Volume

The other special volume is the default volume. This volume is sometimes call the prefix volume. In many applications it is most convenient to have the frequently needed files together in a single volume. If you need to specify their file names frequently, it is tedious to constantly type the volume name or unit number. You can instead tell the File System that when no volume name is specified, the one to use is the default volume. You specify the default volume by using the Filer's Prefix command.

The preceding Volumes display indicates "Prefix is - MYVOL:". This means that MYVOL: is the default volume. The default volume can be specified in two ways. If a colon separator appears with no volume name before it, then the default volume is assumed. If a file name is given with no volume name before it, the default volume is assumed.

Use the Filer's Prefix command to set the default volume name:

```
P
Prefix to what directory?
SYSVOL: Return or Enter
Prefix is SYSVOL:
```

The default volume and the system volume can be the same volume. In fact, except for single-drive configurations, the default configuration you received from the factory has SYSVOL: as both the default and system volume.

If the unit specified by a unit number (e.g., #3:) in a Prefix command does not contain a disc when the Prefix command is executed, then that unit becomes the default volume. That means that any disc in that drive is the default volume, for as long as it is in the drive.

You can also set the default volume name using the **What command** of the Main Commmand Level. The What command is more powerful than the Prefix command because What allows you to specify a new system volume, as well as the name and location of each of the system files (Filer, Editor, Library etc). For further information on the What command, see the "Main Command Level" chapter of this manual.

Files

Information within a blocked volume is further organized into **files**. A file is a collection of related information, having a name by which it may be accessed. Since a volume usually contains many files, within the volume there is also a directory, or "table of contents," telling the name of each file, how big it is, how many sectors it occupies on the disc, and (roughly) what sort of data it contains.

Files are created by computer programs — either system programs (such as the Editor, Filer and Compiler), or user application programs.

For example, when you save a Pascal program written with the Editor, the program is saved with the specified file name in either the current default volume or the specified volume. When that same program is compiled, the object code is stored in another file. When the object code program is executed, it may create more files.

You can use the Filer to list the files in a volume. For instance, to see what is in the default volume of our example system, type:

```
L
```

to invoke the Filer's List Directory command. The Filer responds with:

```
List what directory?
```

To specify the default directory, type in:

[:] [Return] or [Enter]

Assuming the configuration shown on previous pages, you could have done the same job by typing:

SYSVOL: [Return] or [Enter]

The listing of the default volume's directory is shown below.

```
SYSVOL:              Directory type= LIF level 1
created 9-Jan-87 21.13.37  block size=256
changed 9-Jan-87 21.13.37  Storage order
...file name....          # blks  # bytes last chng

TAPEBKUP.CODE            54      13824 28-May-87
FILEINTRO.TEXT           64      16384 28-May-87
FILEINTRO.ASC            73      18688 28-May-87
DATAFILE                 10       2560 28-May-87
FILES shown=4 allocated=4 unallocated=12
BLOCKS (256 bytes) used=201 unused=855 largest space=855
```

File Naming Conventions

The definition of HP Pascal was made to minimize the work of moving Pascal programs from one operating system to another. To do so, string values are used to specify the names of files and certain other information such as passwords and access rights.

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

File Specifications and File Names

There is a difference between a file specification and a file name. A file name is a character string which is the external identifier by which a file is designated in a disc directory. A file specification is a character string which consists of the file name and several other optional items: volume id, directory path, passwords, and size specifier. Not all of these items are allowed by every Directory Access Method or under all circumstances. For instance, passwords are only used with the Shared Resource Management System's hierarchical directory organization, and directory paths only with the SRM and HFS (Hierarchical File System).

Syntax of a File Specification

The syntax of a legal file specification is given by the following diagram:

file_spec ::=[volume_id] [directory_path] file_name ["]"size_spec"]"]

::= volume_id

In this notation, items between square brackets, [and], are optional. Quoted items, such as "[", appear literally. The definition just given means that a file_spec (file specification) may appear in one of two forms. The first form consists of an optional volume id, then an optional directory path, then a file name which is not optional, then an optional size spec. The second form consists just of a volume id, followed by a colon (:).

Examples of the first form are as follows:

```
File_x
A49ZB[10]
#4:LIBRARY.
BOOT:SYSTEM_P
#45:SYSTEM21/FILER
*EDITOR.
```

Examples of the second form are as follows:

```
BOOT:
#3:
*
:
#45:SYSTEM21/TOOLS
#45:
```

Syntax of a Volume Identifier

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

volume_id ::= "#" integer [password] ":"

::= volume_name [password] ":"

In the first case, the integer is a two-digit number from one to fifty; for example, #23: is a volume id. In the second case, the name is a sequence of characters. The length of the name and allowable characters depend on the particular directory organization used by the logical unit. For most "blocked" mass storage devices, the volume name is actually stored on the disc itself so it can be identified whenever it is inserted into a drive. For "unblocked" devices which have no directory, such as printers, the volume name is an arbitrary one supplied by the TABLE configuration program at boot-up time. For HFS volumes, the root or "/" directory may have a real volume id, or a "made-up" volume id that is of the form: "hfs$n$" (where n is an integer value. Note that passwords are not applicable to HFS systems.

Example volume ids of the second form are MYSYS: and PRINTER: Volume_ids may be 6 characters long in LIF directories, 7 characters long in Workstation 1.0 (UCSD-compatible) directories, 14 characters long in HFS directories (other than the root directory which is limited to 6 characters), and 16 characters long in SRM directories. LIF, HFS and SRM allow lower-case, while WS1.0 and serial devices ignore case. WS1.0 converts all characters to uppercase automatically.

In the case of a logical unit connected to an HFS disc or Shared Resource Management System, the volume id takes a special meaning. The notation #5: refers to the current working directory of volume number five; the notation #5:/ refers to the root directory of the SRM or HFS with which volume number five is associated. The current working directory for any SRM or HFS volume is selected by the Filer's Prefix or Unit command, or the Main Command Level's What command.

On the other hand, if the logical unit does not have a hierarchical directory, then **/** is assumed to be a filename. This is the case for all local mass storage devices except when using HFS to access them.

Syntax of a Directory Path (SRM, SRM/UX, & HFS)

Directory paths are only allowed when specifying files on SRM, SRM/UX & HFS logical units. The syntax for a directory path is:

directory_path ::= ["/"] { directory_name [srm_password] "/" }

srm_password ::= "<" word ">"

(SRM passwords are not applicable to HFS or SRM/UX; if SRM syntax is followed, the password will become part of the filename.)

directory_name ::= file_name
     ::= "."
     ::= ".."

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

SRM passwords are sequences of up to 16 characters, which govern the access rights to a file or directory. They are given to a file either at creation time or by use of the Filer's Access command. Note, this does not apply to HFS which has its own security system described in Chapter 5, The Filer under the Hfs command description.

In short, HFS file access permissions recognize an **owner** of a file, a **group** of people who may have privileged access to a file, and all **other** users of the file. The file's **Mode** defines the access permissions of the file, i.e. it allows you to restrict or grant permission, providing you are the owner of the file.

SRM/UX systems employ HFS file access permissions.

Note that a directory path usually doesn't appear by itself; it appears as part of a file specification, with the file name after the directory path. Examples of directory paths are:

/.<PASS1>/	Denotes root, using password "PASS1". (On HFS, this would be a directory under the root directory named ".<PASS1>").
/USERS/ROGER/	Denotes directory ROGER in USERS, which is in root directory.
HERE/THERE/	Denotes directory THERE, found in HERE.
../THERE<PASS2>/	Directory THERE, found in the parent of the current working directory. (On HFS this would be trying to access a directory called "THERE<PASS2>").

A directory path together with a volume id might appear as follows:

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

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

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

That is, the volume password may either immediately precede or follow the colon separator. Volume passwords are not applicable to HFS discs, which have their own security system (discussed near the end of this chapter).

Syntax of File Names

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

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

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

File Types Derived from File Names

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

Suffix	File Type
.ASC	LIF ASCII text file
.TEXT	WS1.0 / UCSD-compatible text file
.CODE	Workstation Pascal object code
.BAD	File covering bad area of disc
.SYSTM	System Boot file
.UX	File of bytes used primarily for data exchange with HP-UX systems
<NONE>	"Data" file assumed when suffix is missing or un-recognized

File Names (LIF DAM)

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

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

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

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

1. Look for a standard suffix (for example, ".ASC").

   a. If a suffix is found, the suffix characters are removed from the name, leaving a trailing period. If this name is longer than 10 characters, including the period, then an error is reported.

   b. If no suffix is found, and the file name contains less than 10 characters, the file is assumed to be a Data file and the name is put into the directory unchanged. If no suffix is found, but the file name is exactly 10 characters in length and the last character is an A, B, C, S, T or U, then an error is reported.

2. If the file is not a Data file and no error has been reported, the dot is replaced by the first letter of the suffix; for instance, the .ASC suffix is replaced by A. If the name is now less than 10 characters long, it is extended to a length of 10 characters by appending underscore characters (_) to the name.

Using this algorithm, we would have the following examples:

File name	Translated name
'A.ASC'	'AA_____'
'charlie'	'charlie '
'123456789.TEXT'	'123456789T'
'GollyGeeeT'	rejected because it would be confused with transformation of 'GollyGeee.TEXT'

The reverse transformation is fairly obvious:

1. If the 10th character is a blank, do nothing. Otherwise, go to step 2.

2. Remove all trailing underscores.

3. Compare the last non-underscore to the first letter of each valid suffix. If a match is found, remove that letter from the file name and append a dot '.' followed by the full suffix.

4. If no suffix match is found, use the original file name.

File Names (Workstation 1.0 DAM)

The Workstation 1.0 (UCSD-compatible) DAM allows file names of up to 15 characters, including the suffix. Any lowercase letters are transformed to uppercase so that 'a.text' and 'A.TEXT' denote the same file.

File Names (SRM DAM)

The SRM itself allows almost any file name, although the Pascal File System removes blanks and control characters from file names.

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

File Names (HFS DAM)

HFS allows filenames of up to 14 characters, including the suffix. Other than the password (which is not applicable to HFS) and the difference in permitted length of filenames, filenames which are valid for SRM are also valid for HFS. Consult the section *Wildcards* later in this chapter before assigning names to new files if you have never used wildcards before.

File Size Specification

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

    size_spec ::= "[" integer "]"

This specification only takes effect if a new file is being created with REWRITE, OPEN, APPEND, or APPEND with OPEN . If the file already exists, the File System tries to make it at least the size specified. The size is ignored for RESET.

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

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

If no size specifier is present when space for a new file is being allocated, the largest free area is assigned to the file, except when using HFS or SRM file systems which allocate no space (SRM and HFS file systems allocate space when needed).

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

For files stored on an HFS disc, the [*] is ignored but a valid size specifier in the format [*integer*] will be accepted and allocated as described above. Note that on HFS, files may have some overhead that is "invisible". For example, if you create file ABC.TEXT[1000] on HFS, the file will consume 1024 blocks, not 1000 blocks, though the usable size will be 512 000 bytes.

Several Directory Organizations Allowed

HP LIF (Logical Interchange Format) is the default directory format used by your Pascal Work-station System. There are many (mutually incompatible) ways to organize files and directories on a disc. LIF is an HP standard disc organization used to transport files among (recently man-ufactured) computers from Hewlett-Packard. The HP Series 200/300 BASIC Language System also supports the LIF directory structure on your Series 200/300 computers. HP-UX provides *lifcp* and other utilities for handling LIF directories.

In addition, your Pascal Workstation understands three other disc directory organizations. The WS1.0 format was the primary disc directory format used by the Pascal 1.0 Workstation File System. Your Pascal Workstation also supports hierarchical directory structures used by the Shared Resouce Management System, and HFS, which is used by HP Series 200/300 HP-UX revisions 5.0 and later. Hierarchical directories and their application to SRM, SRM/UX, and HFS are discussed as a separate topic later in this section. The WS1.0 format is compatible with the widely used UCSD Pascal[1] system.

File Name Suffixes and File Types

Here are five examples of legal file names, although some are too long for LIF or WS1.0 direc-tories:

```
FILEINTRO.ASC
FILEINTRO.TEXT
FILEINTRO.UX
TAPEBKUP.CODE
DATAFILE
```

The first four file names have a suffix. This suffix is part of the file name, so FILEINTRO.ASC, FILEINTRO.TEXT and FILEINTRO.UX are different files. The suffix was appended to specify the file type when the file was created. The file type is stored in the directory along with the file name. Thus, the file type would *not* be changed if you later changed the file name by removing or changing the suffix. You can see the file type of each file by listing the directory using the Filer's Extended Directory List command.

The suffixes recognized by the Pascal 3.2 File System are shown below:

.ASC   These files are HP Logical Interchange Format (LIF) ASCII files. It is intended to be the method for interchanging data files among various HP computers. Information stored in a .ASC file is stored as individual records. Each record has a two-byte length header and contains an even number of bytes. (A pad character is added, if necessary, to make an even number of bytes.)

---

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

TEXT	Text files are the default type of file produced by the Editor. They follow the WS1.0 (UCSD-compatible) format. .TEXT files consist of a 1024-byte header containing Editor environment information, followed by compacted text in 1024-byte pages. Each line of information with more than one leading blank begins with a Data Link Escape control character (DLE, ASCII code 16). This character is followed by a one-byte indentation code; its binary value is calculated by adding 32 to the number of initial blank characters in the line. (Lines with zero or one leading blank merely put the actual line there; there is no DLE/leading blanks compression.) Each line is then terminated by a Carriage-Return control character (ASCII code 13). If a line will not fit at the end of a 1024-byte page, then the whole line is moved to the beginning of the next page, and the remainder of the previous page is padded with ASCII nulls (chr(0)).
.UX	A .UX file is recognized as one which is used primarily for exchange of data with HP-UX systems. Data can be either regular text or binary.
.CODE	A .CODE file is the object code produced by the Compiler, Assembler, or Librarian. This file is also called a library.
.SYSTM	A .SYSTM file is a special file recognized by the Boot ROM as a file containing an operating system (a "system Boot file").
.BAD	A .BAD file is used to permanently cover failed disc sectors. Use the Filer's Make command to make a file of type .BAD over the defective sector of the disc media. This type of file should only be used as an emergency measure. The defective disc should be replaced. For HFS systems, you should also run the file system check utility (HFSCK) which is covered later in this chapter.
<no suffix>	A file whose name at the time of creation does not end in one of these suffixes or ends with an unrecognized suffix is said to be of type DATA. The data may be textual or binary in nature.

The Pascal operating system utility programs (e.g., Editor, Filer) in many circumstances automatically append the appropriate suffix to a file's name. (Note that the only time the Filer appends a suffix is in the Get and Save commands.) For instance, when loading a file into the Editor, just type the file name without a suffix. The Editor knows that in normal circumstances you will want to edit a .TEXT file and will automatically add the suffix. Of course, if you wish or need to specify a suffix, you may. For instance, if you want the Editor to load another type of file, then the correct suffix must be specified. If the file has no suffix, place a period at the end of the file name. The period stops the Editor from adding the .TEXT suffix. If you try to specify a file type that the subsystem can't work with, such as a .CODE file in the Editor, you may get various kinds of undesirable results. In this example, depending on how you access the file and the mass storage device used, you could get an error message reading "Illegal I/O request", a "No workfile found" message, or myriad empty lines.

Automatic suffixing is very convenient. For instance, you might write a program with the Editor and call the output file WORK. The Editor automatically appends .TEXT. When you use the Compiler to compile WORK, the compiler automatically appends .TEXT to the source file name, and .CODE to the output file name. Although there are two files, you only need to remember one name. To execute WORK.CODE, you need only type the following (from the Main Command Level):

[Select] WORK [Return]
or
EXECUTE WORK [Enter]

Suppressing the Suffix

On the other hand, you may wish a file name which has no suffix. You can suppress the automatic appending of a suffix by typing a period as the last character in the file name. For instance, to create a data file with the name AFILE, just tell the Editor to save your file as "**AFILE.**". The period aborts the suffix and makes the type DATA. Likewise, the Librarian and Compiler will automatically append .CODE to file names unless you tell them not to with the period; the type will still be CODE, however.

System programs like the Editor don't have the .CODE suffix. This protects them against accidental destruction by a wildcard purge operation on all .CODE files. If you wish to permanently load a system program into memory with the Permanent command, you must append a dot to the file name. To load the Editor, type:

[P] EDITOR. [Enter]
or
[P] EDITOR. [Return]

Without the period, the system would try to load EDITOR.CODE, which is not what you want.

Translating Files from One Type to Another

Sometimes you may want to translate the contents of a file from one file type to another. For instance, you may have a file of type .TEXT, created by the Editor, and wish to read it with the BASIC system. BASIC understands the LIF ASCII (.ASC) format, but not the .TEXT format. You can use the Filer's Translate command in this situation. A typical dialogue would be:

[T]
Translate what file?
EXAMPLE.TEXT [Return] or [Enter]
Translate to what?
EXAMPLE.ASC [Return] or [Enter]

For Pascal 3.2 and later, the same principle can be applied to convert a .TEXT .ASC, or Data file into a HP-UX compatible file. The dialogue for this could be:

[T]
Translate what file?
EXAMPLE [Return] or [Enter] (Note this is a Data file)
Translate to what?
EXAMPLE.UX [Return] or [Enter]

Note that if you have a file of type .UX which contains tabs and you wish to convert this file to another type, the Translate command expands the tabs into spaces, and the new file will not contain tabs. Eight-column tab stops are presumed during the conversion. The UXTEXT_AM module must be present in the system for this to be possible (as shipped, the UXTEXT_AM is already installed in INITLIB).

For Translate to make sense, the source file must contain data that is textual in nature; attempting to translate a .CODE file, for example, would not make sense.

Translating Text Files to the Printer

Another situation where translation is required is to move a file from a disc volume to the printer. The file may be a .TEXT, .ASC, DATA, or .UX file . The way such files are stored on the disc is not compatible with unblocked devices (such as the printer), so you must use the Translate command. Just type in:

```
T
Translate what file?
WORK.TEXT Return or Enter
Translate to what?
PRINTER: Return or Enter
```

This example illustrates several points. First, in the Filer environment, you must always specify the complete file name including the suffix. Second, to send a file to a device like the printer which has no directory, there is no point in specifying a file name. Just use the volume name. Had you specified a file name after PRINTER: the Filer would have given you an error message.

Wildcards

In the Filer environment, you can specify a particular file or set of files by giving a pattern which identifies the files you want. These patterns include special characters called **wildcards**.

For example, we can use the wildcard = (equal sign) to list a subset of the file using the Filer's List Directory command. From the Filer subsystem, press:

```
L
```

The computer responds with:

```
List what directory?
```

Respond with:

```
FILE= Return or Enter
```

FILE= uses the equal sign wildcard to specify all files whose names begin with FILE and end with any sequence of characters. Using our example system, this command sequence would produce:

```
V12:                 Directory type= LIF level 1
created   9-Aug-86 21.13.37 block size=256
changed 28-Oct-86 15.08.24 Storage order
...file name....      # blks    # bytes  last chng

FILEINTRO.TEXT          64       16384 28-Oct-86
FILEINTRO.ASC           73       18688 28-Oct-86
FILES shown=2 allocated=4 unallocated=12
BLOCKS (256 bytes) used=201 unused=855 largest space=855
```

Notice what happened here. The Filer recognized that the response to the prompt, "List what directory?", specified not just a volume name but a set of files within that volume.

More than one wildcard may appear in a single file specification given to the Filer, allowing you to easily describe some rather complex operations. For instance, you can copy all the files on unit #13 whose names contain the characters INT to the system volume by means of this command sequence:

[F]
Filecopy what file?
#13:=INT= [Return] or [Enter]
Filecopy to what?
*$ [Return] or [Enter]

This example uses the destination wildcard "$", which means "use the same name as the source file had". The command locates each file on unit #13 whose name matches the pattern, and writes a new copy with the same file name on the system volume. Remember that "*" is shorthand for the name of the system volume.

You can use "?" as a wildcard instead of "=". Question mark works like equals, except that for each file whose name matches the specification, the Filer will **ask** if you want to perform the operation. For example, to have the Filer change each file name on the default volume beginning with FILE into a file name beginning with WORK, type in:

[C]
Change what file?
FILE? [Return] or [Enter]
Change to what?
WORK= [Return] or [Enter]

For example, this would turn FILE_ONE.TEXT into WORK_ONE.TEXT. Each time the specification is met, the Filer will present what it has found and ask if the process should be completed for the entry. Answer with [Y] for yes or [N] for no each time you're asked.

File Names to Avoid

The File System won't prevent you from creating file names containing wildcard characters, but you'll be sorry if you do. The Filer will think such file names are wildcard specifications instead of simple file names. For instance, if you created a file called =.TEXT, then used the Filer sequence:

```
R
Remove what file?
=.TEXT Return or Enter
```

the Filer would remove **every** file whose name ends in .TEXT in the default volume!

Should you ever accidentally create a file with a wildcard in its name in volume VOLNAM, you can get rid of it this way:

```
R
Remove what file?
VOLNAM:? Return or Enter
```

This will cause the Filer to offer to remove each file in the directory VOLNAM:. You can then remove the problem file and retain the other files.

Allowable File Names

What file names are allowable depends on the type of directory used on the volume in which the file resides. In other words, the Directory Access Method determines the file name rules. The exact rules for file names are given in the subsequent chapter of this manual called "Programming with Files". Here is a summary of the rules.

It is wise to choose names consisting of alphabetic letters and digits only. If you must use a punctuation mark within a file name, a hyphen, an underscore, or a period is acceptable. Blanks are removed from file names.

In LIF, SRM, SRM/UX, and HFS directories, uppercase and lowercase letters are distinct; "CHARLIE" is not the same file as "Charlie". In WS1.0 directories, lowercase letters in a file name will automatically be converted to uppercase. This exception makes it easier to use wildcards to move files from one type of directory to another. Within HP Workstation Pascal, a typed in lowercase suffix will automatically be transformed into uppercase when writing the file to LIF disc.

Don't Use These Characters

"$", "?", "="	Filer wildcard characters.
"*", ":", "#"	Used in specifying volumes.
"<", ">"	Have special meaning with the Shared Resource Manager.
"/"	Has a special meaning with HFS and SRM.
"[", "]"	Used to specify the size of a file when it is created.
control characters	Control characters are automatically removed from file names.
" "	Blanks are removed from file names.

File Name Length

In LIF directories, file names (without suffix) are limited to 9 characters. If the last character in the file name is **not** an A, B, C, S, or T, then 10 characters can be used. If a suffix is present, up to 9 characters may precede the dot and suffix.

In WS1.0 directories, file names may be up to 15 characters **including** the suffix.

In HFS directories, file names may be up to 14 characters **including** the suffix.

In SRM directories, file names may be up to 16 characters **including** the suffix.

In SRM/UX directories, file names may be up to 14 or 16 characters, depending on the file length allowed in the hosting HP-UX file system.

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.

You may be able to solve this problem by using the Filer's Krunch command. This command consolidates all of the volumes free space by moving all of the files on a volume to the front of the volume. This applies to LIF and WS1.0 volumes **only** and not to SRM or HFS volumes which have different methods of storage.

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 your volume is, use the Filer's Extended Directory List command. This command lists both the files and the fragmented space on the volume.

The SRM and SRM/UX Systems

The concepts presented so far have all been applied to local mass storage devices. The same concepts extend naturally to deal with shared mass storage.

The Shared Resource Management (SRM) and SRM/UX Systems allow several workstations (computers) to be connected into a network that allows sharing of files and resources. This network is controlled by the server, which is a centrally located computer with discs, and possibly some printers and plotters attached. Since the files can now be shared between several users, a new directory structure is needed. Setting up the SRM or SRM/UX systems is not described in this manual; see the SRM or SRM/UX documentation for that information. Configuring your workstation to access an SRM system is described in Chapter 4, "Workstation Startup with Pascal" in the *SRM Software Installation Manual*. Accessing an SRM/UX system is described in Chapter 7 of the *SRM/UX Administrator's Guide*, and in Appendix G of the *Pascal 3.25 User's Guide*.

Hierarchical Directories

The SRM and SRM/UX systems use a hierarchical directory structure to organize its files. This directory structure is a multi-way tree data structure. That is, the first (or top) directory of the structure is called the **root directory**. Subordinate to the root directory are other directories which, in turn, may have further subordinate directories. Each directory may contain files or other directories. When a directory contains **only** files (and no directories), it is called a **leaf directory**. All files can be called **leaf files**. The drawing below shows a hierarchical directory system.
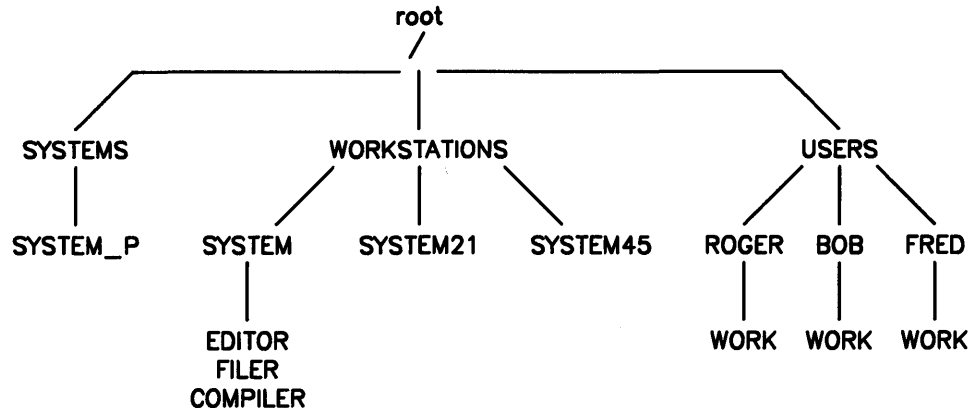
The directory SYSTEMS is a special directory checked by the Boot ROM (version 3.0 or later) for the presence of operating or language systems ("Systm" type files).

The directory USERS (in the above example) has three subordinate directories: ROGER, BOB, and FRED. Each subordinate directory has a single file called WORK. Each file and directory is uniquely specified by the list of directories from the root to the file. That means several files of the same file name can exist without confusion if they are in different locations in the directory structure.

To save space, the Filer's Duplicate Link command can be used to link a file into a directory other than its original location. This allows you to have access to a file, such as the Compiler or Editor, without making an extra, unnecessary copy. See the Filer chapter of this manual for more information.

Once a duplicate link has been set up, if the file is purged from the directory, what happens to the link? Only the directory from which the file was purged loses access to the file. All other directories with links to the file can still find it. The disc space allocated to the file is only reclaimed when no directories have links to it.

Notation

Hierarchical directories are a simple concept, but we need some specialized words and notation to talk about them.

The directory at the top of the hierarchy is called the "root" directory. If we want to refer to a file or directory which is immediately under the root, for instance WORKSTATIONS in the illustration above, you would specify:

/WORKSTATIONS

This is read as "slash WORKSTATIONS" or "stroke WORKSTATIONS". The / indicates the root directory.

To go further down the hierarchy, for instance to SYSTEM under WORKSTATIONS, specify:

/WORKSTATIONS/SYSTEM

and for another level yet, specify:

/WORKSTATIONS/SYSTEM/COMPILER

As you can see, to specify a file, the list of directories from either the root directory or the current working directory to the target file must be specified. The directories in the list are delimited with /'s. More information about the current working directory is given later in this section.

Such a sequence of strokes and directory names is called a directory path, since it indicates the path one must follow down the hierarchy to get to a particular file.

SRM Units and Volumes

A workstation connected to an SRM or SRM/UX server normally has units #5: and #45: set up for server access. The use of two units is in keeping with the idea that there are usually two special volumes (the system volume and the default volume) through which most file accesses occur.

If the workstation has no local mass storage, unit #45: will automatically be configured to be the system volume and unit number #5: will be available for use as the default volume. If there is local mass storage, the system volume can be any volume you desire. To set these volumes, use the What command from the Main Command Prompt.

Here is how the Filer's Volumes display might look right after booting up a workstation connected to the SRM and having no local mass storage:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  5 # MYSRM:
  6   PRINTER:
 45 * SYSTEM21:
Prefix is - SRM:
```

You can see that the system starts out with #5: as the default volume and #45: as the system volume.

Where do the names MYSRM: and SYSTEM21: come from? They are actually the names of particular directories in the SRM's hierarchy. In this example, the name of the SRM volume is MYSRM:, and the workstation we are using is at node address 21. Since there is a directory SYSTEM21, it is selected as the system volume. All of this selecting is done by the TABLE program as it automatically configures the system each time you boot.

If you need to specify the SRM volume's password, you can do it by using this syntax:

MYSRM<password>:filepath

SRM/UX systems do not support volume or file passwords.

Moving Up and Down the Hierarchy

It would be tedious to type a directory path every time you wanted to access a file. To avoid this, you can specify the **current working directory** using the Filer's Unit Directory command. The current working directory can be used as the "root" to specify subordinate files.

```
 U 
Set unit to what directory?
#5:/USERS/ROGER  Return  or  Enter 
```

Once you have done this, unit #5: is associated with the directory named ROGER which is subordinate to the USERS directory in the root. It's similar to inserting a disc called ROGER: in a disc drive. If you now command the Filer with:

```
 L 
List what directory?
ROGER:  Return  or  Enter 
```

it will list all the files in volume ROGER: which is directory /USERS/ROGER. You could also use the sequence:

```
L
List what directory?
#5: Return or Enter
```

since directory ROGER was installed in #5: by the Filer's Unit Directory command.

Suppose that under ROGER is another directory named MYSTUFF which contains more files. To list the files in MYSTUFF, use the sequence:

```
L
List what directory?
ROGER:MYSTUFF Return or Enter
```

The Filer will realize that MYSTUFF under volume ROGER is itself a directory, and list its contents. If MYSTUFF were not a directory, it would simply be listed as a file in directory ROGER.

You can move the current working directory still farther down the hierarchy in the obvious way. For instance, to make MYSTUFF the current directory of #5:

```
U
Set unit to what directory?
#5:MYSTUFF Return or Enter
```

There was no need to specify the entire directory path from the root because MYSTUFF was already accessible as a file within volume ROGER.

A special notation is provided to move up the hierarchy. Two periods can be used to denote the superior, or "parent," directory of a file. For instance, after moving down to MYSTUFF, unit #5: could be moved back up to the parent directory ROGER by:

```
U
Set unit to what directory?
#5:.. Return or Enter
```

To go up two levels, use the double-period twice, separated by a slash:

```
U
Set unit to what directory?
#5:../.. Return or Enter
```

This can be executed all the way up to the root directory. Of course, if you want to get all the way to the top, it is easier to go there directly, using a stroke as the "name" of the root directory. For instance, while #5: is assigned to MYSTUFF you could list all the files in the root directory with the command sequence:

```
L
List what directory?
#5:/ Return or Enter
```

Default Volume vs. Current Working Directory

The current working directory concept is different from the default volume concept. Specifying a current working directory is like installing a disc into a drive. Specifying a default volume simply tells the File System what volume name to use when none is specified with a file name.

The two concepts can come together in the Filer's Prefix command. For instance, typing:

[P]
```
Prefix to what directory?
#5:/USERS/BIG_USER [Return] or [Enter]
```

has two effects since #5: is an SRM unit. The current working directory of #5: is set to /USERS/BIG_USER, and the default volume name is set to BIG_USER. If we now type:

[U]
```
Set unit to what directory?
#5:.. [Return] or [Enter]
```

the current working directory of #5: becomes USERS (the parent of BIG_USER). However, the default volume name is still BIG_USER. So this command sequence:

[L]
```
List what directory?
: [Return] or [Enter]
```

will fail with the message that BIG_USER is not on-line!

The same sort of mistake is commonly made with the system volume. Suppose the current working directory of #45: is SYSTEM21, and the COMPILER, EDITOR and other system files are under SYSTEMS. If the current working directory of #45: is changed, the Operating System won't be able to find the system programs since it thinks of them as SYSTEM21:COMPILER and so on. If this happens, you need to get into the Filer and restore the current working directory of #45: . How can you do so if the Filer is no longer on-line? You will need to eXecute the Filer by name, specifying a path all the way down from the root to wherever it is:

[Select] (**EXECUTE**)
```
Execute what file?
#45:/WORKSTATIONS/SYSTEMS/FILER. [Return] or [Enter]
```

Note the dot after the Filer's name. You don't want the system to append .CODE in this case.

SRM and SRM/UX Concurrent File Access

The SRM and SRM/UX systems also provide the capability of several users concurrently accessing "shared" files. For further information on this capability, see the "Programming with Files" chapter in Volume II of this manual.

SRM Access Rights

You can restrict the use of SRM files with a special "access right" password-protection scheme. For further information on this capability, see the "Programming with Files" chapter in Volume II of this manual. Also see the Access command description in the "Filer" chapter for more information on assigning and removing SRM passwords with the Filer. The SRM "access right" scheme is not supported on SRM/UX.

The Workstation Hierarchical File System

This section will give you an overview of the terms and methodology associated with Workstation HFS.

General Information

The Workstation Hierarchical File System (HFS) is provided with HP Workstation Pascal 3.2 and later versions. It is recommended that this be used in preference to other file systems provided.

For those who have SRM systems, the principle of storing and retrieving data in a file system hierarchy is nothing new. The ability to create an HFS on local discs will improve the organization and overview of your files, and, through this, hopefully your productivity.

If your experience so far only involves the LIF and WS1.0 disc directory formats, you can think of an HFS as being a natural progression from your existing directory format. All files are stored in a tree structure, the first or top directory in the structure is called the **root directory**. Every directory may contain both files and further directories. However, the way in which the information is physically written onto the storage media is different to LIF and WS1.0. The files are non-contiguous. Data exchange with BASIC 5.0 and HP-UX 5.1 (and later) systems, and the ability to boot any of these operating systems from the same HFS disc is possible.

As this file system permits data exchange with BASIC and HP-UX, a new file type has been introduced which is compatible with the HP-UX environment. This '.UX' file is described earlier in this chapter and in the *Programming with Files* chapter in Volume II of this manual.

To create an HFS on an initialized disc, you must first run the MKHFS utility supplied with the system. To boot a system from an HFS disc, you will also need to use the OSINSTALL utility to install SYSTEM_P file on your disc and copy other files into a special directory. These utilities are covered in detail in Chapter 21, "HFS Set-Up and Utilities", in Volume II of this manual.

To operate in an "HFS environment", you need to either execute the HFS_DAM module and then TABLE, or incorporate the HFS_DAM module into INITLIB so that it is executed on booting the system.

Terminology

HFS and the utility programs supplied to support the file system use some new terminology. The following information can also be found in the Glossary at the end of this manual. It is not really intended that you simply read this; it is more as a local source of reference to you. Many of these new terms are used extensively in the file system checking utility HFSCK. Here is an explanation of the words you may not recognize:

Blocks and Fragments
Files stored on an HFS disc are non-contiguous, which means that the information within the file is spread over various areas of the disc. These are called blocks. Of course, the data may not be equivalent in size to an exact number of blocks. To avoid wasting a lot of space on an HFS disc, the file system also works with fragments, which can simply be looked upon as smaller blocks. Blocks and fragments have fixed sizes which are defined when the file system is created. The default sizes are 8192 and 1024 bytes respectively.

Boot area
A boot area is reserved on an HFS disc during the installation of the file system by the MKHFS utility. Information in the boot area is only used for booting, and the boot area resides on the first 8K bytes of the disc. The boot area contains a LIF volume header.

Bytes per Inode
This specifies the ratio of data bytes to inodes. It reflects the average size you expect your files and directories to be; the default value is 2048, implying an average file size of 2 kilobytes.

Cylinder
One or more vertical collections of tracks in a disc or disc pack.

Cylinder Groups
Cylinders are accessed in groups by HFS to minimize the head movement when accessing files. Although the files are non-contiguous on the disc, cylinder group information is used to keep the pieces close together when possible, thus speeding access.

Inode
Among other information pertinent to a file on an HFS disc, an inode contains one or more pointers which specify where a file's data blocks are located. As a file may be split into a number of sections, the inode holds the information concerning the location of each section, enabling the file system to find the complete file. Each inode is numbered for reference by HFSCK when a problem occurs. The inode number for a file can also be seen when a Filer extended listing is produced of a directory on an HFS disc.

Superblock
This is a block on an HFS disc which contains global information about the file system as a whole and which enables the file system to operate with the disc; for example, how much disc space is still available and where it is located. It does not contain information about individual files stored on the disc. The superblock is created at the same time as the file system and is replicated into each cylinder group to provide redundancy.

Workstation Header
(wsheader)

Since the HFS directory structure has no place in which to keep information such as file type (e.g., .ASC, .CODE) or logical end-of-file, the HFS driver allocates some space at the start of the data area of your file to keep track of these things. Normally you will never see this area of the file since the HFS driver automatically skips over it. The size of the area is 512 bytes for each file created by the Pascal Workstation on an HFS disc, except for .UX type files and directories. Note that HP-UX tools do not skip this area and the user must decide how to handle the wsheader. For this reason, .UX files are recommended for file interchange between HP-UX and the Pascal Workstation. BASIC 5.0 creates and uses compatible wsheaders.

Using HFS

Notation

The drawing below shows a typical HFS directory structure.
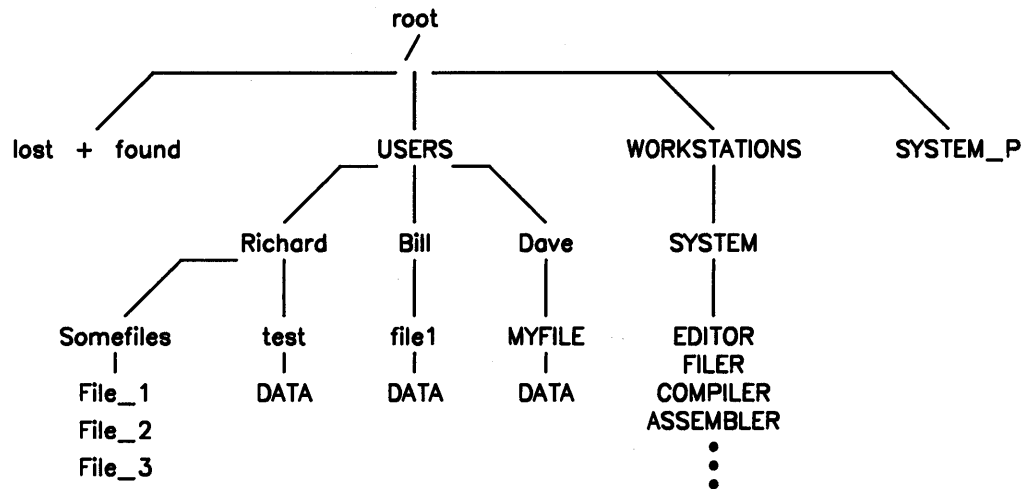
Hierarchical directories are a simple concept, but we need some specialized words and notation to talk about them.

The directory at the top of the hierarchy is called the "root" directory. In the diagram, lost+found, USERS, and WORKSTATIONS are directories within the root directory, and SYSTEM_P is a simple boot file also in the root directory.

The directory lost+found is a special directory in the root directory of every HFS disc **and must not be deleted**. It is used by the HFSCK utility, which is described in the HFS Utilities chapter.

To refer to a file or directory which is within the root directory, for instance USERS in the illustration above, you would specify:

   /USERS

This is read as "slash USERS" or "stroke USERS". The / indicates the root directory.

To go further down the hierarchy, for instance to Richard under USERS, specify:

    /USERS/Richard

and for another level yet, specify:

    /USERS/Richard/Somefiles

If your current working directory was Richard, then to access File_1 you would only need to type:

    Somefiles/File_1

As you can see, to specify a file, the list of directories from either the root directory or the current working directory to the target file must be specified. The directories in the list are delimited with the character "/". More information about the current working directory is presented later in this section.

Such a sequence of strokes and directory names is called a directory path, since it indicates the path one must follow down the hierarchy to get to a particular file.

HFS Units and Volumes

Before you can use your disc with the above notation, the disc must have the HFS file system installed on it. Delivered with Pascal 3.2 and later versions is a utility program which provides this function, called MKHFS. For more information on this utility and instructions on how to use it, consult the HFS Utilities chapter in the Pascal Workstation System manual Volume II.

Here is how the Filer's Volumes display might look right after booting up a workstation connected to a local HFS disc:

```
Volumes on-line:
    1   CONSOLE:
    2   SYSTERM:
    6   PRINTER:
   11 # hfs11:
   46 * SYSTEM:
 Prefix is -hfs11:
```

HFS uses the normal unit numbers you have become accustomed to, with two exceptions. If, at boot time, the system discovers the directory /WORKSTATIONS/SYSTEM on the boot disc, and that disc is HFS, it will assign that disc to be the system volume and allocate unit "#46" to it. Unit #46 will be set to the path /WORKSTATIONS/SYSTEM. This is the situation in the above listing of volumes given by the Filer. Flexible or floppy discs, due to their being removable and replaceable, will not have #46 assigned to them.

HFS and Flexible Disc Unit Numbers

For flexible discs, the system cannot predict what file system will be on the inserted media. Another problem is that HFS discs usually have a small LIF header at the "front" of the disc for booting purposes. The system will assign flexible disc units in pairs, for example #3 paired with #43, and #4 paired with #44. The unit at #3 "understands" LIF discs, and the LIF area of HFS discs. The unit number at #43 only understands the HFS part of the disc (if it is an HFS disc). The other pairs work in a similar fashion. To protect your HFS discs, be sure not to do any operations with unit #3 if unit #43 indicates an HFS disc. Possible problems may included loss of data or the inability to boot from the HFS 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. The "Adding Modules to INITLIB" section of Chapter 18 discusses the AMIGO and other driver modules.

For flexible disc drives, the system must be able to distinguish between discs inserted into the drive which are LIF format, and those which are HFS format. To inform the system which type of disc currently occupies the drive, the following rule is applied. If the drive occupies the unit numbers #3 and #43, an HFS disc inserted in this drive can *only* be accessed by using #43. Similarly, a LIF disc inserted in this drive can only be accessed by #3. This rule is applicable to all unit number "pairs" assigned to flexible disc drives, i.e. #3 and #43, #4 and #44, #6 and #46, #7 and #47, #8 and #48, and #9 and #49.

Depending on the format of the disc being used, the system will use a particular Directory Access Method or DAM. For LIF discs a LIF DAM is used and similarly for HFS an HFS DAM is used. For more information on DAMs, consult the *Using Alternate DAMs* section in the *Workstation System Manual, Volume II.*

Moving Up and Down the Hierarchy

It would be tedious to type a directory path every time you wanted to access a file. To avoid this, you can specify the **current working directory** using the Filer's Unit Directory command. The current working directory can be used as the starting point to specify subordinate files.

```
U
```
Set unit to what directory?
#11:/USERS/Richard/Somefiles `Return` or `Enter`

Once you have done this, the unit (#11: in this example) is associated with the directory named Somefiles which is subordinate to the Richard and USERS directories. It's similar to inserting a disc called Somefiles: in a disc drive.

If you now command the Filer with:

`L`
List what directory?
Somefiles: `Return` or `Enter`

it will list all the files in subdirectory Somefiles. You could also use the sequence:

`L`
List what directory?
#11: `Return` or `Enter`

since directory Somefiles was installed in #11: by the Filer's Unit Directory command.

Suppose that under Somefiles is another directory named DIR_1 which contains more files. To list the files in DIR_1, use the sequence:

`L`
List what directory?
Systemfiles:DIR_1 `Return` or `Enter`

The Filer will realize that DIR_1 in volume Somefiles is itself a directory, and list its contents. If DIR_1 were not a directory, it would simply be listed as a file in Somefiles.

You can move the current working directory still farther down the hierarchy in the obvious way. For instance to make DIR_1 the current directory of #11:

`U`
Set unit to what directory?
#11:DIR_1 `Return` or `Enter`

There was no need to specify the entire directory path from the root, because DIR_1 was already accessible as a file within volume Somefiles.

A special notation is provided to move up the hierarchy. Two periods can be used to denote the superior, or "parent," directory of a file. For instance, after moving down to DIR_1, unit #11: could be moved back up to the parent directory Somefiles by:

`U`
Set unit to what directory?
#11:.. `Return` or `Enter`

To go up two levels, use the double-period twice, separated by a slash:

`U`
Set unit to what directory?
#11:../.. `Return` or `Enter`

This can be executed all the way up to the root directory. Of course, if you want to get all the way to the top, it is easier to go there directly, using a stroke as the "name" of the root directory. For example, while #11: is assigned to DIR_1 you could list all the files in the root directory with the command sequence:

```
L
```
List what directory?
#11:/ `Return` or `Enter`

Note that HFS root directories can be named and renamed but the maximum length is six characters, the stroke being a convenient generic shorthand. If you have never given your root directory a name, the system will have given it a default name. The name consists of four or five characters with the format "hfs*nn*", where *nn* normally corresponds to the unit number.

Default Volume vs. Current Working Directory
The current working directory concept is different from the default volume concept. Specifying a current working directory is like installing a disc into a drive. Specifying a default volume simply tells the File System what volume name to use when none is specified with a file name.

The two concepts can come together in the Filer's Prefix command. For instance, typing:

```
P
```
Prefix to what directory?
#11:/USERS/BIG_USER `Return` or `Enter`

has two effects since #11: is an HFS unit. The current working directory of #11: is set to /USERS/BIG_USER, and the default volume name is set to BIG_USER. If we now type:

```
U
```
Set unit to what directory?
#11:.. `Return` or `Enter`

the current working directory of #11: becomes USERS (the parent of BIG_USER). However, the default volume name is still BIG_USER. So this command sequence:

```
L
```
List what directory?
: `Return` or `Enter`

will fail with the message that BIG_USER is not on-line!

Booting from an HFS Disc
To be able to boot an operating system from an HFS disc you should follow the detailed instructions given in the HFS Utilities chapter in the OSINSTALL section.

When booting from a non-removable HFS disc, units #11 and #46 will be assigned to the HFS disc as the Prefix and System volumes respectively.

Security of Files and Directories

The security of files and directories on HFS discs is controlled by a combination of **user class** and **permission type** "values".

User Class

There are three classes of user who have access to files and directories:

Owner   The owner of a file is normally the person who created it, however it is possible to transfer ownership. Pascal Workstation and BASIC are "persons" for this purpose, and are users 17 and 18, respectively.

Group   Where there are users working as a group, files (and directories, which are merely a special type of file) can be assigned a group ownership. All members of the group, except the person who is the owner of the file, share the same permissions. Pascal 3.2 and BASIC 5.0 are both in group number nine (9).

Other   This class of user includes anyone who has access to the system but is neither the owner nor a member of the group.

Permission Type

There are three types of permission for each file or directory created:

Read (r)   A read permission granted for a file allows a user to look at or load into the computer the contents of a file. For directories, the read permission allows a user to catalog the directory.

Write (w)   Users who have write permission for a file can change the contents of that file. For directories, write permission allows the user to create and delete files within the directory. Note that if you have write permission for a directory, but not for a particular file in the directory, you may still delete that file.

Execute (x)   Execute permission is meaningless for files in a Pascal Workstation environment, but for directories it permits a search through a directory for filenames or further directories, without the user being able to catalog the directory. If you do not have execute permission on the directory in which the file exists, you will not be able to open the file for reading or writing. Execute permissions for files are used by HP-UX so it is advisable not to remove this permission on a file of type Hp-ux.

You can find out whether you are the owner of a file by looking at the "directory info" column of an Extended directory listing. It could look like this for a file:

```
...directory info...

664m       17u       9g
```

We will ignore the "664m" for now. If there is a code "17u" then you are the owner (or "user") of the file. If the number is not 17 then you are not the owner and the *owner access rights* are not applicable to you for the file. This means that for this file you are either a member of a specified group or simply someone who has access to the system. You see the code "9g" to the right of the owner code. If the number is not 9 then you are not a member of a group which

has access to this file. Assuming that for the file chosen, you are neither the owner or a group member, then you are classified as "Other". The number 9 is a reserved number specifying a group for the Pascal and BASIC environments. In BASIC the owner number is 18.

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 the above example of a "directory info" column of an Extended directory listing, the code on the left ("664m") 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.

Workstation HFS and HP-UX

This section attempts to clarify the level of compatibility with HP-UX the Workstation HFS provides. If you do not have HP-UX and do not intend sharing data with anyone who has, you can skip this section.

Compatibility

Any compatibility between Pascal Workstation HFS and HP-UX starts with version 5.1 of the HP-UX operating system running on the Series 200/300. The extent of this compatibility is discussed below.

Where HFS Provides Compatibility

This file system enables you to share discs with HP-UX and therefore provides you with the capability of booting either operating system from the same disc. It also permits sharing of textual and binary data.

Where HFS is not Compatible

The HFS does not permit the sharing of object code or applications. It does not allow you to run Pascal under HP-UX or vice-versa, nor does it mean that all the HP-UX file types are supported. The fact that Pascal and HP-UX can coexist on one disc does not mean that this disc becomes a multi-accessible disc. Workstation Pascal does not provide the same *login* type of security when operating with an HFS disc. This means that all Pascal users are the same user to the HP-UX system (user 17).

File Handling

As discussed earlier, the implementation of HFS and the implications of compatibility with HP-UX require a new file type for Workstation Pascal. This new file type is called "Hp-ux" (.UX suffix).

HFS Listing Information

For HFS discs, the Filer's Extended directory listing contains some very useful information:

HFS Extended Listing

```
hfs11:      Directory type= HFS 755     17      9
changed 15-Mar-87  14.24.54 block size=1024
 Storage order
...file name....    # blks    # bytes    start blk ....last change... extension1
                    type   t-code ..directory info... ....create date... extension2

lost+found               8       8192          3 15-Mar-87 2. 4.14           2
                  Dir     3 d755m     17u   9g                              -1
TOP                      1         96         64 8-May-86  8. 9.53           3
                  Dir     3 d755m     17u   9g                              -1
just_a_file              8       7168          4 15-Aug-86 6.10.24           1
                  Data -5622 644m     17u   9g                            7618
FILES shown=3 allocated=3 unallocated=18497
BLOCKS (1024 bytes) used=17 unused=21726
```

At the top, the listing prints the volume name (hfs11:), and the directory type (HFS). The numbers following the directory type indicate the access permission for the directory (755), the owner (17 — the "Pascal" user), and the group (9 — the "Workstation" group).

Note

If you do not have the HFS_DAM installed, an HFS disc will appear to the workstation as a LIF disc. This is very dangerous since writing to an HFS disc as if it were a LIF disc will destroy data stored under the HFS disc.

Two lines are used to list the information for each file in the directory.

- The first of the two lines gives the file name, the size in blocks, the size in bytes, the i-node number (in place of the start block), the last change date and time, and the number of links to the file (in the extension1 field).

- The second of the two lines gives the type (Dir, Text, etc.), the type-code (numerical value for the type), the directory info (details below), and type specific data (in the extension2 field).

The "Directory info" column shows the public access rights and the ownership of the HFS file. The field is split into three subfields. The first is the mode (as in 644m), the second the user or owner (as in 17u), and the third the group (as in 9g). If the file is a "special" file such as a directory, the file mode will begin with a letter (e.g. d for directory). Other letters include p, n, s, etc. Other than directories, most special files are not supported by the Pascal Workstation and should not be manipulated. If the t-code indicates a zero (0), the Pascal Workstation should not be used to manipulate the file, as it is a special file.

The access and ownership information for HFS files and directories is given in the form of octal numbers (numbers of base 8). In the listing above, the two directories, lost+found and TOP, have access rights defined in the Directory info column by d755m. The "d" shows that this file is in fact a directory and the octal number is 755. For the file called just_a_file, the access rights are given by the number 644. For a detailed explanation of these codes and the other information in the "directory info" column see the Extended directory and Hfs commands in the "Filer" chapter of this manual.

Extensions

The extension1 and extension2 fields provide system-dependent file information. For files and directories on HFS discs, extension1 is the number of links. For files appearing only in one directory, this field contains 1. For a directory, there will always be at least 2 links (i.e. the second link appears within the directory as a link to itself).

For most files, the extension2 field usually contains -1 unless the file has a Workstation header (WSheader).

The Editor   **4**

Introduction

This chapter introduces the features of the Workstation Pascal Editor. The Editor enables you to create, change, store and retrieve text files, which may be either programs or other textual documents. Like other parts of the system, the Editor has built-in reminders (prompts) and uses single keystroke commands.

The Editor is a cursor-based screen editor. The cursor, normally shown on the screen as a underline character, shows where subsequent characters will be inserted into the text. You can rapidly access any part of the text file by moving the cursor to the desired location.

The programs and documents created by the Editor are usually stored as text (.TEXT - suffix) files, but can also be stored as ASCII (.ASC - suffix), HP-UX compatible text files (.UX - suffix) or data (no suffix) files.

This chapter has four main sections.

- The first two sections, "Entering the Editor" and "A Sample Editor Session", demonstrate how to enter and use the Editor by leading you through writing a short Pascal program.

- The next section, "A Closer Look", presents more detailed information about the Editor.

- The last section, "Editor Commands", contains an overview or summary of all the Editor commands, a glossary of terms, and a semantic and syntactic description of each command, in alphabetical order.

Once familiar with the Editor, you can use the overview/summary of the Editor commands for quick reference.

Note that this manual assumes that you are using an HP 46020 or 46021 style of keyboard. If you are using one of the 98203 style keyboards, the Editor's prompt line will be slightly different. It is also assumed that your screen is 80 columns by 24 lines. Other size displays may cause the text to appear slightly different from what is shown here.

Entering the Editor

It is assumed that the Pascal System is already "up and running" The following prompt should appear on the top line of your screen:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

(If the screen does not display this prompt, refer to the *Pascal User's Guide* for loading instructions.)

This prompt tells you that you are at the system's Main Command Level — the level from which all the Pascal subsystems (Compiler, Editor, Filer, etc.) are entered. Entry into any subsystem is accomplished by typing the uppercase character of the name of the subsystem you wish to enter.

Note

If you have a system workfile (created in a previous Editor session or in the Filer subsystem), first go into the Filer and use the Save, New, and Quit commands to store and release the workfile. Then exit the Filer subsystem.

When the system is delivered to you, the Editor is on the disc labeled "ACCESS:" and is named:

```
ACCESS:EDITOR
```

Now press the ⌈E⌋ key. You can use uppercase or lowercase: the computer treats both exactly the same while at the Main Command Level. If the Editor code file is on-line, the screen clears and displays:

```
Loading 'ACCESS:EDITOR'
```

If you copy the Editor code file to another disc, which has a different volume name, you should tell the operating system where to look for the Editor. (See the What command in Chapter 1)

Creating a Text File

When you enter the Editor, the following prompt is displayed.

```
Editor [Rev.  3.2 15-Feb-87]

Copyright 1987 Hewlett-Packard Company.
         All rights reserved.

No workfile found.
File? (<ret> for new file, <stop> exits)
:
```

This tells you that you are entering the Editor without a system workfile and requests a file name. Respond by pressing the [Return] or [ENTER] key to instruct the Editor to create a new text file for your use. The file will be named later when leaving the Editor.

The Editor can also be directly entered from the Compiler subsystem. This is covered in Compiler chapter.

The Editor Prompt

The screen clears again and displays the Editor prompt on the top line:

```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?
```

You are now in the Pascal Editor with a new file. The Editor prompt shows the most common commands used in the Editor. This is called a "prompt" because it prompts you to take some action, i.e., give the Editor a command.

The first character of the prompt line (> or <) indicates the direction of cursor movement (i.e., the way the cursor moves when [Tab], [Return] or [Enter] keys, or the space bar is pressed). When the > character is displayed, the cursor will move "forward" in the text. Similarly, when the < character is displayed, the cursor will move "backwards" in text. Pressing [>], [.], or [+] will set forward direction, while [<], [,], or [-] will set reverse direction.

The character indicates the direction that searches take place in the Find and Replace commands, also the Delete and Page commands.

The prompt line shows a partial list of commands available in the Editor. To see the rest of the commands, type [?]. The screen shows the Editor's alternate prompt:

```
>Edit: Margin  Page  Set environment  Verify ?          [3.2]
```

This alternate prompt also shows the revision number of the Editor enclosed in brackets. Type [?] again and the main Editor prompt reappears.

All of the commands in the Editor are initiated by typing a single key corresponding to the first character of the command shown in the Editor prompt. Again it does not matter whether the character is uppercase or lowercase — the Editor accepts either one. Now that you are in the Editor and understand something about the Editor prompt, let us begin the sample Editor session.

A Sample Editor Session

Feel free to skim this section if you are familiar with screen-oriented editors. You may even prefer to try out the Editor commands on your own. If you choose to experiment with the Editor commands, do not use any files you cannot afford to lose.

If you are still reading, step through the following examples on your machine. Doing the examples will help you learn faster than just reading about them.

Creating Text

The most direct way to generate text is with the Insert command. Initiate the Insert command by pressing ⏹I and the screen responds with the following prompt:

```
>Insert: Text <bs>, <clr ln>    [<sel> accepts, <esc> escapes]
```

While in the Insert command, any of the regular character-entry keys (the main keyboard) or the numeric pad keys (on the right) may be used. With a few exceptions, using the key clusters on the top of your keyboard or ⏹CTRL key sequences is not advised. (Most of these keys generate a question mark "?" while in the Insert command. Others have results which may surprise you). Use ⏹CTRL key sequences only if you are working with Stream files. (See Chapter 1 for details on the use of Stream files). The exceptions are the cursor control keys, ⏹Back space, ⏹Clear line, **ANY CHAR** and **DUMP ALPHA** (which sends a copy of the screen image to your printer).

Note

The Editor does not permit control characters to be placed in the text file. Attempting to edit a file containing control characters (created by some other system) can lead to unexpected results.

Let's start typing in a Pascal program now. Press ⏹Return or ⏹Enter and then type the text shown in the display below. If you make a mistake, use ⏹Back space to move the cursor backward and then type the correction. Prompts in the Editor always show actual key options in the form of a key abbreviation shown between < and >. For example, **<sel>** indicates the ⏹Select key (on HP 46020 and 46021 keyboards), while **<exc>** indicates the ⏹EXECUTE key (on 98203 keyboards).

The word "binary" is misspelled in the display; leave it that way for now.

Notice that when you press ⟨Return⟩ or ⟨Enter⟩ after typing the first line, the cursor returns to column one (the column that the P in PROGRAM is in). To type the second line, use ⟨Tab⟩ to indent the comment enclosed in the braces. The next time you press ⟨Return⟩ or ⟨Enter⟩ the cursor automatically returns to the indented position created in the previous line. This indenting feature proves handy when writing Pascal programs as it adds visual clarity to the code.

```
>Insert: Text <bs>, <clr ln>     [<sel> accepts, <esc> escapes]

PROGRAM Binery_search(INPUT,OUTPUT);
          {This program does a binery search
          on an array of characters to find a
          "key" character input by the user.}
```

The display above shows what your screen should look like after the first few lines are typed. To move the cursor back to column one for the next line, press and hold ⟨Back space⟩. The keyboard automatically "repeats" any key that remains pressed.

```
>Insert: Text <bs>, <clr ln>     [<sel> accepts, <esc> escapes]

PROGRAM Binery_search(INPUT,OUTPUT);
          {This program does a binery search
          on an array of characters to find a
          "key" character input by the user.}

VAR                done : BOOLEAN;
                    key : CHAR;
                  alpha : ARRAY [1..26] of CHAR;
       loop, top, mid, btm : INTEGER;
```

When your screen looks like the display above, press ⟨Select⟩ (**EXECUTE**) to complete the insertion. The screen displays the Editor prompt along with the text you inserted. Next we will save this program fragment on the disc and then return to create more text.

Storing your File and Returning to the Editor

This section shows how to save a file on a disc and then return to the Editor. It is a good idea to do this periodically when writing and editing large text files. Although power outages occur infrequently, it can be devastating to lose an entire session of work. Occasional updating of your file secures your work against this possibility.

Press ⟨Q⟩ to initiate the Quit command. The screen clears and displays:

```
>Quit:
     Update the workfile and leave
     Exit without updating
     Return to the editor without updating
     Write to a file name and return
```

Before typing anything, find the disc labeled DOC: and insert it in your disc drive in place of the disc labeled ACCESS:. Now press ⌷W⌷ and the screen displays:

```
>Quit:
Name of output file (<ent> to return) -->
```

The prompt is requesting a file specification. Respond by typing DOC:BINSEARCH followed by ⌷Return⌷ or ⌷Enter⌷. The screen now displays:

```
>Quit:
Writing..
Your file is 275 bytes long.
Exit from or Return to the editor?
```

The exact number of bytes may differ with what is indicated in the line above.

Now press ⌷R⌷. The screen fills with your text and the cursor is positioned where it was when you initiated the Quit command.

Copying Text from Other Files

The Insert command is the most common way of creating text but other commands are available. The Copy command allows you to copy specified text from another file.

On the DOC: disc is a text file called BINDOC.TEXT which you are going to copy into your current text file. Position the cursor by pressing ⌷J⌷ ⌷E⌷ . This command sequence moves the cursor to the end of your text file. (More on the Jump command later). Now press ⌷C⌷ and your screen displays:

```
>Copy: Buffer File <esc>
```

The Buffer option is demonstrated along with the Delete command later in this section. Now press ⌷F⌷ (to Copy from a File) and the new prompt appears:

```
>Copy: File[marker,marker] ?
```

The system is requesting a file specification. Type DOC:BINDOC and press ⌷Return⌷ or ⌷Enter⌷. The .TEXT part of the file name does not have to be typed; it is automatically supplied by the Editor. The volume name , DOC:, had to be specified because otherwise the Editor would look for the file on ACCESS:, the default volume. See Chapter 2 for further information on the system default volume.

The entire file DOC:BINDOC.TEXT has been copied into your current text file in memory. The copy always occurs at the cursor position. This is why you moved the cursor to the end of the file before the copy. The screen now appears as follows:

```
PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR            done : BOOLEAN;
                key : CHAR;
              alpha : ARRAY [1..26] of CHAR;
   loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN         {This is the actual binery search}
 mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

To Copy only part of a file, a beginning and ending marker are specified. These markers must have been previously set in the text file being copied. (See the Set command in the "Editor Commands" section of this chapter for further information on setting markers). Now that you have your screen full of text, let's look at the general pattern of leaving an Editor command and some ways to move the cursor.

Confirming or Aborting Commands

The [Select] (**EXECUTE**) key tells the Editor to accept all of the insertions or changes you have made in the text file. The cursor remains where it was when you pressed [Select]. Conversely, pressing [ESC] or holding down the [Shift] key while pressing [Select] (shown as [Shift]-[Select]) tells the Editor to ignore all of the changes made since initiating the command and leaves the cursor where it was when the command was initiated. Both key sequences ([Select] and [Shift]-[Select]) return you to the main Editor prompt.

The changes are stored in the computer's internal read/write memory but are not made permanent on a mass storage medium until you exit the Editor and use one of the options that writes the information to a file.

Not all commands let you abort changes with [Shift]-[Select] and not all require [Select] for acceptance. For instance, the Copy from buffer command is accomplished by simply pressing [C] [B]. The text is copied and the Editor's prompt appears with no other action on your part. The specifics of how each command uses these keys is discussed as each command is presented.

Moving the Cursor

Now that you have some text on the screen, experiment with positioning the cursor. The four arrow keys, the [Return] or [Enter] key, the [Tab] key, the space bar and the cursor wheel (also called the knob) all move the cursor.

Note

The cursor wheel is optional on some Series 300 computer keyboards. If you have one, you will need to execute the HPHIL and MOUSE modules before the wheel will operate correctly.

The wheel normally moves the cursor left or right depending on which direction you turn it. If you turn the wheel while holding down the [Shift] key (or after pressing the knob button), the cursor moves up or down while remaining in the same column position.

An integer in the range 1 to 9999 can be used as a "repeat factor" before all of the cursor control keys and some of the Editor commands. (Repeat factors must be in the range 1 to 4095 for use with the [Tab] key). The result will be the same as if you had pressed the key that many times. For instance, typing the number 42 and then pressing the space bar causes the cursor to move 42 characters in the current direction. The [/] key can also be used as a repeat factor; it means "as many as possible". For example, pressing [/], [→] moves the cursor to the end of the file regardless of the length of the file.

The Jump Command offers another means of cursor positioning. Press [J] and the top of your screen displays:

```
>JUMP: Begin End Marker <esc>

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
         on an array of characters to find a
         "key" character input by the user.}
```

Typing [B] causes the cursor to jump to the beginning of the file, in this case directly above the P in PROGRAM, and the Editor's main prompt reappears. Now press [J] then [E] and the cursor moves to the end of your text file as shown in this partial display:

```
    IF key = alpha[mid] THEN done:= TRUE
    ELSE IF key < alpha[mid] THEN top:=mid
    ELSE btm:=mid;
    IF top=btm THEN BEGIN
                    done:=TRUE;  mid:= -1;
                END;
 END;
 IF mid > 0 THEN
  WRITELN('Key -',key,'- is in position ',mid:2)
 ELSE WRITELN('key - ',key,' - was not found');
 END.
```

You can also Jump to previously set markers (see the Set command in the "Editor Commands" section) by typing ☐J☐ ☐M☐ followed by a marker name.

The beginning and end of a file are simply the first and last characters in the current text file. In this case, the position directly above the P in PROGRAM and the space following the final word END. are the first and last characters, respectively. The Editor adjusts these internal pointers automatically as the text file is changed.

The Page command lets you move through a file one screen (that is 23, 24, or 47 lines depending on the dispay size) at a time. It is roughly equivalent to using a repeat factor of 23, 23, or 47 with ☐↑☐ or ☐↓☐ depending on the direction shown in the prompt. If the cursor is not at the end of the file, press ☐J☐ ☐E☐. Now type < to change from the forward to the backward direction and press ☐P☐ (for Page). The top half of your screen now looks like:

```
<Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap  ?

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
    loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
```

Notice that the cursor is positioned near the VAR declaration in the program which is 23 lines from the end of the file. Since the cursor movement direction is still backward, type > to change it to forward. The Page command is especially handy when moving through a large file.

Deleting Text

Now position the cursor under the first brace ({) character on the second line of the program and press ☐D☐. This initiates the Delete command. Moving the cursor removes text from the file. To restore the text, use any cursor control key which moves the cursor back over the area where text has been removed. The ☐Back space☐ key and the cursor wheel work well for this.

Upon pressing ☐D☐, the screen displays:

```
>Delete < > <Moving commands> [<sel> delete, <esc> aborts]

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
    loop, top, mid, btm : INTEGER;
```

First make sure the direction is forward (>) as shown above and then type **4** followed by ⌈Return⌉ or ⌈Enter⌉. This uses a repeat factor and moves the cursor 4 lines, deleting text as it goes. (The deleted text is temporarily stored in the copy buffer). Now press ⌈Select⌉ (**EXECUTE**) and the screen displays:

```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
    loop, top, mid, btm : INTEGER;
```

Before typing any other keys or moving the cursor, press ⌈C⌉ then ⌈B⌉. This takes the information stored in the copy buffer and copies it into the text file beginning at the current position of the cursor. Since the Delete command just filled the buffer with the text that was removed, the Copy from Buffer command simply returns the screen to its state before the Delete command was entered.

The top of the screen should now display:

```
>Edit: Adjst  Cpy  Dlete  Find  Insrt  Jmp  Rplace  Quit  Xchng  Zap ?

PROGRAM Binery_search(INPUT,OUTPUT);
          {This program does a binery search
          on an array of characters to find a
          "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
    loop, top, mid, btm : INTEGER;
```

Recovering Deleted Text

As the example shows, even if you complete the Delete command using ⌈Select⌉ (**EXECUTE**) instead of ⌈Shift⌉-⌈Select⌉, you can still change your mind and recover that text using the Copy (from buffer) command. Take care not to depend on this too heavily as there are other commands which alter the contents of the buffer. None of the cursor movements alter the buffer in any way.

Moving and Duplicating Text

The sequence of the Delete and Copy (from buffer) commands provide a convenient way of moving text to different parts of the file. For instance, in the operation just completed above, any of the cursor control keys could have been used to reposition the cursor after the deletion occurred and before the Copy from the buffer was executed.

The buffer is "filled" with the text affected by the Delete command and by the Insert and Zap commands. Doing a Copy from buffer sequence does not change the contents of the buffer. This feature lets you to copy the same text in numerous places.

Whether the Delete command was completed with the [Select] (**EXECUTE**) or [Shift]-[Select] sequence makes no difference to the copy buffer. What this means in practical terms is that the Delete command allows you to fill the buffer without affecting your original text.

So if you want to duplicate the text instead of moving it to a different location, use the sequence:

1. Press [D] to initiate the Delete command.
2. Cause some cursor movement. After pressing [D], moving the cursor deletes text and stores it in the copy buffer.
3. Press [Shift]-[Select] to recover the text that was just deleted.
4. Reposition the cursor to where you want to duplicate the text.
5. Press [C] [B] to duplicate the text at the new cursor position.

Changing and Altering Text

Mistakes or necessary changes in a program or text file are not always obvious when creating text. The Editor features commands which allow you to go back and make changes when needed. These are the eXchange and Replace commands and the Delete/Insert command sequence. These will be demonstrated by making corrections to the sample program text.

Press [J] [B] to move the cursor to the beginning of the file and then type [5] and press [R] to initiate the Replace command. The prompt at the top of the screen appears:

```
>Repl[5]: L V <targ><sub>=>
```

Press [L] and [V] to tell the Editor that you are going to give it a Literal string and that you want to operate in the Verify mode. A Literal string may occur as a word or as part of a word. The alternative is a Token string which must occur as a word. The Verify mode makes the changes one at a time after asking you if you want this occurrence replaced. Now type:

```
/inery//inary/
```

The slashes were used to delimit the target and substitution strings; however, any non-alphanumeric or non-control characters can be used as delimiters. (In fact, that is necessary when the slash is part of the search string or replacement string.) Notice also that "inery" is specified instead of "binery". This is because two occurrences of the word are "Binery". The two words are unequal.

After you type the final delimiter, the screen clears and displays:

```
>Repl[5]: L V <targ><sub>=>

PROGRAM Binery_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}
```

The cursor is positioned behind the first occurrence of the string **inery**. Now press ⬚R⬚ and watch what happens:

```
>Repl[4]: <sh_exc> aborts, R replaces, ' ' doesn't

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
  loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;  {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {This is the actual binery search}
 mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

The first string `inery` has been replaced with `inary`, the cursor is now positioned behind the second occurrence of the target string and the prompt shows that you can make four more replacements. Press the space bar (represented by ' ' in the prompt) to leave the string unchanged and the screen now displays:

```
>Repl[3]: <sh_exc> aborts, R replaces, ' ' doesn't

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR              done : BOOLEAN;
                  key : CHAR;
                alpha : ARRAY [1..26] of CHAR;
   loop, top, mid, btm : INTEGER;

BEGIN {Binery_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {This is the actual binery search}
```

The cursor is now behind the occurrence of **Binery** following the **BEGIN** statement. Press ⌐R⌐ to replace this one and then press it again to replace the last occurrence of **binery**. The screen now displays:

```
>ERROR: Pattern not found. <space> continues.

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}

VAR             done : BOOLEAN;
                 key : CHAR;
               alpha : ARRAY [1..26] of CHAR;
   loop, top, mid, btm : INTEGER;

BEGIN {Binary_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {This is the actual binery search}
 mid:= ROUND((top + btm)/2);
   IF key = alpha[mid] THEN done:= TRUE
   ELSE IF key < alpha[mid] THEN top:=mid
   ELSE btm:=mid;
   IF top=btm THEN BEGIN
```

The prompt at the top of the screen tells you that the Editor could not find any more occurrences of the specified string in the file. The cursor is positioned at the final occurrence of the string but it has not yet been changed. Press the space bar and the Editor prompt reappears, the final occurrence of the string gets replaced and the cursor remains at the same place on the screen.

To correct the spelling of **binery** (which was intentionally skipped), use the eXchange command. Move the cursor to the **e** in **binery** in the second line of your program. Now press ⌐X⌐ and the screen shows:

```
>Xchnge: Text <bs> <esc> aborts <sel> accepts

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binery search
        on an array of characters to find a
        "key" character input by the user.}
```

Type the letter **a** and then press ⌐Select⌐ (**EXECUTE**). Pressing ⌐Select⌐ confirms changes made in eXchange and returns the Editor prompt. That's all there is to the eXchange command.

You should always position the cursor before initiating eXchange because this command cannot cross line boundaries; you can only make eXchanges on the line where the cursor is located and only after the cursor.

The eXchange command is handy but the combination of the Insert and Delete commands is often a more effective way to change text. For example, to clarify the program by adding comments, position the cursor at the comment following the second BEGIN, press ⎡D⎤, and press ⎡Tab⎤ once. The screen displays:

```
>Delete: < > <Moving commands> [<sel> deletes, <esc> aborts]

PROGRAM Binary_search(INPUT,OUTPUT);
        {This program does a binary search
        on an array of characters to find a
        "key" character input by the user.}

VAR                 done : BOOLEAN;
                     key : CHAR;
                   alpha : ARRAY [1..26] of CHAR;
     loop, top, mid, btm : INTEGER;

BEGIN {Binary_search}
 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {        the actual binary search}
 mid:= ROUND((top + btm)/2);
    IF key = alpha[mid] THEN done:= TRUE
    ELSE IF key < alpha[mid] THEN top:=mid
    ELSE btm:=mid;
    IF top=btm THEN BEGIN
```

Using a combination of ⎡Tab⎤ and the space bar, delete everything between the two brackets and press ⎡Select⎤. Part of the screen looks like:

```
             .
             .
             .
WHILE NOT done DO
BEGIN         {}
  mid:= ROUND((top + btm)/2);
             .
             .
             .
```

Press ⬚Ⅰ to initiate the Insert command and notice how a space is opened between the brackets. Insertions always occur directly in front of the cursor's position when Insert is initiated. Now type in the text shown below and then press [Select] to complete the insertion.

```
                   .
                   .
                   .
    WHILE NOT done DO
    BEGIN           {This routine compares key to
                    middle.  A new top or bottom is chosen
                    and a new middle computed.  The loop
                    continues until either key = middle or
                    the array is exhausted.}
        mid:= ROUND((top + btm)/2);
                   .
                   .
                   .
```

Formatting Text

The Pascal Editor allows you to format text with the Adjust and Margin commands. Text is also formatted by inserting or deleting blanks where needed.

The Editor's Adjust command provides a means of shifting the starting column of a line of text left or right in the file. This command helps make your Pascal programs and other text more readable. To increase the clarity of our sample program, move the cursor to the word **mid** following the second **BEGIN** statement in the program. Press ⬚A and the Adjust prompt appears:

```
Adjust: Ljust Rjust Center <arrow keys> [<sel> to leave]
```

Experiment with the Adjust command by pressing ⬚L, ⬚R, or ⬚C. These options move text to the left, right, or center. The values used to shift the text are the Left and Right margins of the environment (discussed below). Any of the cursor arrow keys as well as [Back space] and the cursor wheel can be used to Adjust text. Now return the line to its original position and press **EXECUTE**. Repeat factors are available for use with the Adjust command so that many lines of text can be shifted at one time.

Note

Think twice before using Adjust with large repeat factors. This is because [Shift]-[Select], which usually aborts all changes made by a command, is not available for exiting the Adjust command. Therefore, to recover the original format of your text, you would have to Adjust it again.

Now that the line is in its original place, press ⬚A (to initiate Adjust), type ⬚3 ⬚► (to indent the text three spaces to the right), and then type ⬚6 and press ⬚▼. Watch what happens: the cursor moves down six lines and shifts each line three spaces to the right. Thus, the Adjust command is useful for indenting entire blocks of text in a Pascal program.

The screen now looks like this:

```
Adjust: Ljust Rjust Center <arrow keys> [<sel> to leave]

 done:=FALSE;  btm:=0;  top:=26;   {initialize}
 FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
 WRITELN('Type uppercase character for a key');
 READ(key);  WRITELN;
 WHILE NOT done DO
 BEGIN          {This routine compares key to
        middle.  A new top or bottom is chosen
        and a new middle computed.  The loop
        continues until either key = middle or
        the array is exhausted.}
     mid:= ROUND((top + btm)/2);
     IF key = alpha[mid] THEN done:= TRUE
     ELSE IF key < alpha[mid] THEN top:=mid
     ELSE btm:=mid;
     IF top=btm THEN BEGIN
                         done:=TRUE;  mid:= -1;
                     END;
 END;
 IF mid > 0 THEN
  WRITELN('Key -',key,'- is in position ',mid:2)
 ELSE WRITELN('key - ',key,' - was not found');
 END.
```

Press ⌐Select⌐ to terminate the Adjust command. If you wish to make adjustments in other parts of your text file, exit the Adjust command using ⌐Select⌐ before moving the cursor from one area to another. Otherwise you may make unwanted adjustments to your text.

The Margin command lets you margin and fill your text according to a predefined "environment". Margin operates on the paragraph where the cursor is located when ⌐M⌐ is pressed. A paragraph is any text delimited by any combination of blank lines, lines whose first non-blank character is the "command character" (see the Set environment command in the section "Editor Commands"), or the beginning or end of a file. The Margin command is executed completely by pressing ⌐M⌐; no parameters or options are available.

Entering the Editor without a workfile or a named file (as you did earlier in this session) automatically sets (or defaults) the environment to the "program" environment. This environment is optimized for writing programs. When the Editor is entered with either the name of a "TEXT" type file or a workfile, the environment associated with that file is the current environment.

You can alter the environment at any time using the Set (Environment) command. Once you have altered or redefined the environment and saved a text file on mass storage, that environment is stored along with the text file and is used whenever the Editor is entered with that file.

Since you entered the Editor without a file, your current environment is the Editor's "program" environment (the default supplied by the system). The Filling option of this environment is set to False (which disables the Margin command) so, if you press ⬚M⬚, the screen displays:

```
ERROR: Wrong environment <space> continues.
```

If Filling had been set True (with Auto-indent False), the Margin command would fill and Margin your program like this:

```
PROGRAM Binary_search(INPUT,OUTPUT); {This program does a binary
search on an array of characters to find a "key" character input by
the user.} VAR done : BOOLEAN; key : CHAR; alpha : ARRAY [1..26] of
CHAR; loop, top, mid, btm : INTEGER; BEGIN {Binary_search}
done:=FALSE; btm:=0; top:=26; {initialize} FOR loop:=1 TO top DO
alpha[loop]:=CHR(loop+64); WRITELN('Type uppercase character for a
key'); READ(key); WRITELN; WHILE NOT done DO BEGIN {This routine
compares key to middle. A new top or bottom is chosen and a new
middle computed. The loop continues until either key = middle or
the array is exhausted.} mid:= ROUND((top + btm)/2); IF key =
alpha[mid] THEN done:= TRUE ELSE IF key < alpha[mid] THEN top:=mid
ELSE btm:=mid; IF top=btm THEN BEGIN done:=TRUE; mid:= -1; END;
END; IF mid > 0 THEN WRITELN('Key -',key,'- is in position ',mid:2)
ELSE WRITELN('key - ',key,' - was not found'); END.
```

The previous display gives you some idea of how important it is to know what your environment settings are before using the Margin command. The only recovery from this operation is to use a combination of the Adjust and Insert commands to rebuild the text. If you have a copy of the original file available, you can exit the Editor without updating the file and reenter it with the old copy.

Note

The Insert command has effects similar to those of the Margin command when the Filling option of the environment is set to True and Auto-indent is False. Any time you do an Insert and confirm the operation by pressing ⬚Select⬚, both the inserted text and all the text that follows the insertion in that same paragraph are automatically margined.

The Margin takes place according to the Left and Right margin settings of the environment. If you begin an insertion and are not sure of the environment settings, press ⬚Shift⬚-⬚Select⬚ to abort the Insert command. This way, even if Filling is True, your text will not be margined. Then press ⬚S⬚ ⬚E⬚ to look at the environment settings.

When writing programs, your use of the environment and the Margin command will probably be more limited than when writing other kinds of text. To see how the program environment is configured, press ⎡S⎤ ⎡E⎤ (for Set Environment). The screen displays the current environment:

```
>Environment: {options} <sel> or <sp> leaves
      Auto indent   True
      Filling       False
      Left margin   0
      Right margin  78
      Para margin   5
      Command ch    ^
      Token def     True
      Ignore case   False
      Zap markers
      275 bytes used, 348909 available.

      Patterns:
        <target>= 'inery', <subst>=  'inary'

      Markers:
          TOP             FIX

      File DOC:BINSEARCH.TEXT
      Date Created: 10-11-82   Last Used:  1-11-87
```

Press the space bar to exit the environment display and the Editor prompt reappears along with your text. The cursor is at the position it was when the Set command was entered.

Exiting the Editor and Saving the File

Now that you have finished writing and editing the program, exit the Editor by pressing ⎡Q⎤ (for Quit). Make sure that the disc named **DOC:** is in the same disc drive you have been using. The screen displays:

```
>Quit:
    Update the workfile and leave
    Exit without updating
    Return to the editor without updating
    Write to a file name and return
    Save as file new file DOC:BINSEARCH.TEXT
    Overwrite as file DOC:BINSEARCH.TEXT
```

Respond by pressing ⎡S⎤ for Save. If you are on an SRM system, you would use the Overwrite option. The Overwrite option allows all duplicate links and passwords to a text file to remain intact after a file has been modified. More information on these options is given in the command reference under the Quit command.

```
>Quit:
Writing..
Your file is 1009 bytes long.
Exit from or Return to the editor?
```

Your program text has been written to your disc and is accessible under the name BINSEARCH.TEXT on the volume DOC:.

If you are creating a file for use with another language system, such as BASIC or HPL, the file should be stored as an ASCII type file on a disc with a LIF directory. To accomplish that, use the Write option and name the file:

DOC:BINSEARCH.ASC

On a LIF directory, the Pascal system codes all the file names that end in a standard suffix. The coding scheme removes the period, appends the first character of the suffix to the file name, and pads the file name to ten characters with "_" (underscore characters). This allows you to specify file names up to 15 characters. They are encoded to the maximum ten characters for the LIF directory. The file system encodes the above file name to:

BINSEARCHA

In this case, the first character of the suffix is the tenth character so no "_" characters were added. This coding mechanism is invisible as long as you are using the Pascal system. When you catalogue your disc with other language systems, the coded version of the file name is observed.

To create a file which HP-UX commands can process (e.g. *vi*), the file should be created with a .UX suffix (for example, #11:BINSEARCH.UX). While it is preferable that the file be written to an HFS disc, HP-UX supports utilities that can access LIF discs and an SRM.

Making a Backup Copy

The most direct way to make a backup copy of your file is to press ⌊R⌋ (to return to the Editor) and then press ⌊Q⌋ (to initiate the Quit command). Each time you Quit the editor, you can make another copy of the file currently in the Editor.

Press ⌊W⌋ for the Write option, type in a name for your backup copy such as DOC:BINBKP and press ⌊Return⌋ or ⌊Enter⌋. If you have another disc handy, replace the DOC: volume with it, specify the name of the new volume along with a file name and press ⌊Return⌋ or ⌊Enter⌋. Remember the nine character limit for file names on LIF discs. After pressing ⌊Return⌋ or ⌊Enter⌋, the screen displays:

```
>Quit:
Writing..
Your file is 1009 bytes long.
Exit from or Return to the editor?
```

There are now two identical files on your disc(s) of the binary search program. Now press ⌊E⌋ (for the Exit option) and you will be returned to the Main Command Prompt:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

All the Editor commands covered here are explained in further detail in the "Editor Commands" section. Less commonly used commands not presented in this sample session can also be found there.

A Closer Look

This section contains details about how the Editor works and includes information on the cursor, the screen, memory and file sizes and how the Editor allocates space for text files on a storage medium. The section also presents information on using workfiles in the Editor, on Stream files and on I/O errors that may occur when entering and exiting the Editor.

The Cursor

The cursor (the underline symbol on the screen) is a reference point for all of the Editor's commands. The action associated with most commands occurs at the cursor position. Commands that perform actions on lines or paragraphs act upon the line or paragraph where the cursor is currently located.

You have complete control over the cursor through the arrow keys, the Tab and Return or Enter keys, the space bar, the mouse, and the cursor wheel. (The cursor wheel is also called the knob). The cursor's position on the screen determines where the Editor will act upon the text.

The Anchor

You can also use the Zap command to delete text. With this command, all text between the current cursor position and the "anchor" is deleted. The anchor is set at the position of the latest Adjust, Find, Insert, or Replace command. (You can also find the position by pressing = .)

If more than a 80 characters of text are to be deleted, you will be warned as follows: WARNING! Zap more than 80 characters? (Y/N) Press Y to confirm the Zap operation; press N (or space bar, etc.) to abort the Zap.

The Screen as a Window into a File

Text files are often too large to be shown all at once on the computer's screen (CRT), so the Editor uses the screen as a "window" which shows a portion of a file. Depending on which machine you have, your CRT can display lines of text that are either 49, 79, or 127 characters long while in the Editor. If a line's length is greater than your display area, the Editor puts an exclamation point (!) in the last column to inform you that the entire line is not shown.

The screen is capable of displaying more lines than the Editor uses. The top line is reserved for the system's prompt and the bottom line is reserved for the "type ahead" line. The type ahead line displays any characters input from the keyboard which have not yet been processed by the system. One other item is displayed in the lower right corner of the screen. This is a system status display or "run light." The System or User status of the softkeys is next to the run light.

The screen generally displays the cursor and the text surrounding it. (The Set environment command is an exception to this). This means that you can move the window up and down through your text file by moving the cursor. Whether the text is on or off the screen, it resides in the computer's read/write memory and is easily accessed using either the cursor control keys or the various editing commands which reposition the window. When an Editor command operates on a portion of text it displays that text on the screen.

Memory and File Sizes

When the Editor is entered, the current text file is loaded into the computer's read/write memory. All changes that occur to a text file (including text creation) take place in this memory and only become permanent when the Editor is exited and the contents of the text file are written from memory to mass storage (such as a disc).

The maximum size of the text files that can be accessed or created by the Editor depend on the memory configuration of your system. This size can easily be determined using the Set (environment) command. The two environment headings, "bytes used" and "available", should be added together. The sum equals the maximum size (in bytes) of the text files which can be handled by the Editor.

If your text file approaches the maximum size while you are doing an insertion, the Editor displays the following message:

>ERROR: Finish the insertion <space> continues.

This tells you that you are nearing the Editor's memory limits. If, after finishing the insertion, you attempt to initiate the Insert command again, the Editor informs you:

>ERROR: No room to insert. <space> continues.

Here is procedure to help you work around the Editor memory limits (whatever they may be on your machine):

1. Set a marker toward the end of your original file (to be used later).

2. Quit the Editor, Save the original file, and Exit the Editor completely (in order to re-enter with a new file).

3. Re-enter the Editor and press [Return] or [Enter] (to create a new file).

4. Using the Copy from file command, specify your original file and your marker as follows: FILENAME[MARK,] and press [Return] or [Enter]. (The name of your file and marker will be different; this just shows you the general form.) Notice that a second marker was not specified so that the copy takes place from the marker's location to the end of the original file.

5. Now, press [J] then [E] (to Jump to the end of your new file).

6. Press [I] (to initiate the Insert command).

Now you can continue inserting your text in your new file without too much loss of continuity. You may want to go back to your original file and delete the text that was copied into your new file so that it will not exist in both files.

Structure of Text Files

The Editor can read and write four types of files. The predominant file type is TEXT. The others are DATA, Hp-ux (i.e. HP-UX compatible), and ASCII. TEXT files contain ASCII characters and are structured in a particular way.

In every TEXT type file, the first two blocks (or 1024 bytes) are reserved for information about the environment settings, the locations of markers in the file and other information the Editor needs to work with that file. Since the Editor allocates mass storage in two block increments, TEXT files always contain an even number of blocks. Also, because the Editor reserves the first two blocks for file information, a file with only a single character will take up 4 blocks (2048 bytes) of storage space on a mass storage medium.

It is possible to create a text file that is of type DATA (not of type TEXT). To do that, Quit the Editor, select the Write option, and specify a file name followed by a period at the end of the file name. The Editor does not append .TEXT to the file name, and the file is stored as type DATA. (Note that the environment information is lost in this case.) See the Filer chapter for further information on file types.

If you want to access a DATA file with the Editor, you must specify the file name followed by a period when entering the Editor. If you do not use the trailing period, the Editor appends .TEXT to the name you type in and looks for a file with that name in the specified (or default) volume.

For example, suppose when exiting the Editor you answer the prompt for a file name with DUX. (a name ending with a period). The Editor saves the file with the name DUX (it strips off the period) and does not append the .TEXT suffix. If you enter the Editor and want that file, you must specify DUX. (with trailing period). If you instead specify DUX (i.e., leave out the period), the Editor appends .TEXT to the name you typed and looks for a file with the name DUX.TEXT. It may even find a file with that name, but it will be a different file than the one saved by specifying DUX.. This also applies to HP-UX compatible files (.UX suffix).

You can also create a LIF ASCII file by appending .ASC to the file name. ASCII files are created by writing to a file whose name ends in the suffix:

.ASC

ASCII files are structured differently. ASCII files on LIF discs are compatible with the BASIC and HPL language systems that run on your computer. When writing ASCII files, the Editor's environment information is also lost.

Finally, you can also create an Hp-ux type file by appending .UX to the file name. Hp-ux files are created by writing to a file whose name ends in the suffix:

.UX

Hp-ux files are structured differently. They are compatible with the BASIC and Series 300 HP-UX 5.1 and later Workstations. When writing Hp-ux files, the Editor's environment information is also lost.

Using Workfiles in the Editor

A workfile in the Editor is used as a "scratchpad" version of a text file. The workfile is useful because it is the default file in the Editor (as well as in many of the Pascal subsystems). The workfile is stored on the current system volume (usually a disc). The File System chapter contains information about using workfiles in all the subsystems; only Editor-related workfiles are covered here.

There are two ways to enter the Editor: from the Main Command Level or from the Compiler subsystem (after the Compiler finds an error in the text file it is compiling). When entering from the Compiler, the text file being used is automatically read in. When entering the Editor from the main level, the system automatically looks for a workfile and, if it finds one, reads the contents of the file into the computer's memory. If the Editor does not find a workfile, it prompts you for a file name.

Exiting the Editor (using the Quit command) gives you the option of Updating the workfile. If the Editor was entered with a workfile (or if the Update option was used earlier in the same editing session), the Editor writes the contents of the text file in memory to the file called *WORK.TEXT on the system volume. When you are through with all your editing, it is a good idea to enter the Filer subsystem and Save the workfile.

Stream Files and the ANYCHAR Key

Stream files can be created by the Editor to simulate a "batch" mode in which the computer executes the commands in the Stream file as if they were coming from the keyboard. The **ANYCHAR** key is useful in this regard. It can be used to generate characters which may not otherwise be attainable by regular keystrokes. For further information on the **ANYCHAR** key and Stream files, see the *Pascal User's Guide*.

I/O Errors (Entering and Exiting the Editor)

There are two general types of errors that can occur when entering the Editor. The first type of error is generated by the system when it cannot find the volume or file which you specified. The solution to this is to make sure that the proper volume is on-line. This type of error also occurs when a workfile exists but the Editor cannot find it because the volume containing that file is no longer on-line. When the Editor encounters this situation, it informs you that the workfile has been "lost" and then prompts you for a file name.

The second type of error possible while entering the Editor is a memory overflow condition. This happens if the text file being read was created on a machine with more memory available than the machine currently being used. Note that this condition is met if you use the Permanent command (at the Main Command Level -- see the Overview chapter) to load something into memory that was not there when you created the text file. Your machine now has less available memory, so the space for text files is smaller.

When a memory overflow occurs while reading in the file, the Editor lets you continue the entry process even though the entire file has not been read into memory. However, upon exiting the Editor, the Save option is no longer available. This safeguard keeps you from accidentally overwriting your original file.

When exiting the Editor, a number of different errors are possible. If the Editor detects an error while writing the contents of the text in the computer's memory to a mass storage medium, it will display a self-explanatory error message.

Editor Commands

This section contains a brief overview and summary of all the Editor commands and a complete alphabetized description of the syntax and semantics of all the Pascal Editor commands and options.

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 next pressed.
Insert	Inserts text in front of the current cursor location.
Replace	Replace the specified target string with the specified substitute string.
eXchange	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.
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.

Some Series 300 computers have a knob built into the keyboard and some have a "knob box" that attaches in-line with the keyboard. Both need the MOUSE and HPHIL modules to be installed before they will operate. The "knob box" has a switch on the side that toggles the direction of the scroll similar to the Shift key but the switch does not need to be held down.

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 screenful of text from the current location.

Command Syntax and Semantics

The Editor commands are presented in alphabetical order and described in a variety of formats to make them more useful to you. Each command's explanation includes: the command's name, a brief functional description, a diagram showing its legal syntax, the command's prompt (if any), and a discussion of using the command. Each of the command's options are also covered, and some descriptions include examples to show the proper use of these options.

Alphabetical Listing of Editor Commands

Adjust
Copy
Delete
Equals (=)
Find
Insert
Jump
Margin
Page
Quit
Replace
Set
Verify
eXchange
Zap

Adjust

Adjust horizontally shifts the starting column of one or more lines of text.
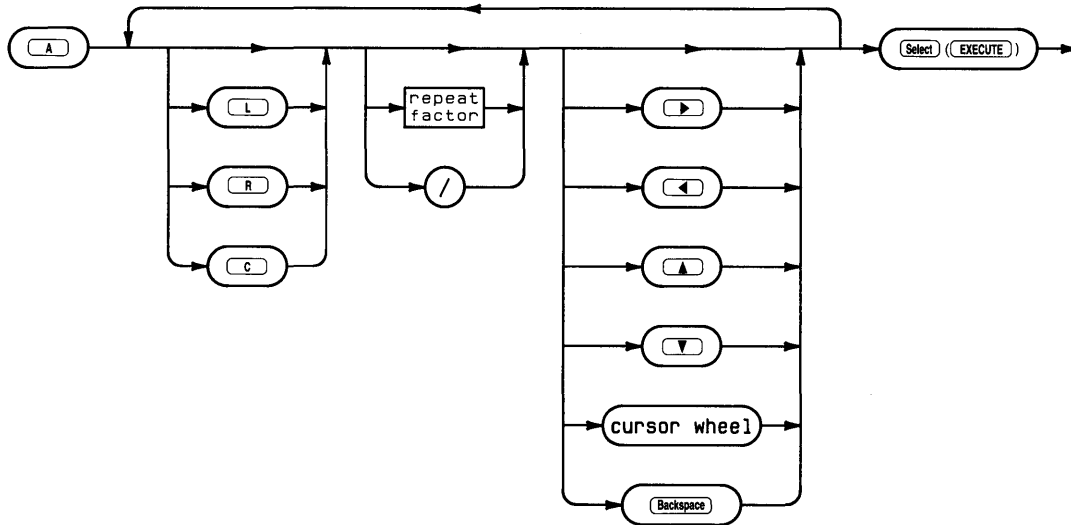
Item	Description	Range
repeat factor	integer numeric constant	1 thru 9999

Semantics

The Adjust prompt:

```
>Adjust: Ljust Rjust Center <arrow keys> [<sel> to leave]
```

The Adjust command provides a means of formatting text and enables you to make text more readable. Adjust uses the line position of the cursor when the command is entered as a starting point. A line-oriented command, Adjust lets you shift an entire line of text to the left or right using the ▶, ◀, Back space, or cursor wheel. Repeat factors can be used with these keys to shift the text. For example, pressing 7 ▶ results in the line of text shifting 7 spaces to the right.

Pressing $\boxed{\text{A}}$ (for Adjust) and then $\boxed{\text{L}}$, $\boxed{\text{R}}$ or $\boxed{\text{C}}$ moves the line to the left margin, right margin or centers the line between the two margins. The margins used by these options are the Right and Left margins currently set in the environment (see Set command).

Typing a repeat factor and $\boxed{\text{▲}}$ or $\boxed{\text{▼}}$ causes that number of lines to be adjusted the same amount as the accumulated adjustments at that point. The slash (/) functions as an infinite repeat factor and can be used with $\boxed{\text{▲}}$ and $\boxed{\text{▼}}$. It causes adjustments to be made from the current line to either the beginning or the end of the text file, respectively. For example, pressing $\boxed{\text{C}}$ $\boxed{/}$ $\boxed{\text{▼}}$ causes all the text between the current cursor position and the end of the file to be Centered according to the current margins.

Note

Take care when using large repeat factors or the slash (/) character to adjust text, because the effects of the Adjust command cannot be aborted. Whatever adjustments are made become permanent unless the Adjust command is used again.

Adjust also sets the anchor used by the Zap command. Pressing = (the Equals command) moves the cursor to the position of the last Adjust unless the anchor has been reset by either a Find, Insert, or Replace command.

Leave the Adjust command by pressing $\boxed{\text{Select}}$ (**EXECUTE**). The system stores the adjusted text in the computer's memory and the Editor prompt reappears.

Copy

Copy inserts text from a specified file or from the copy buffer.
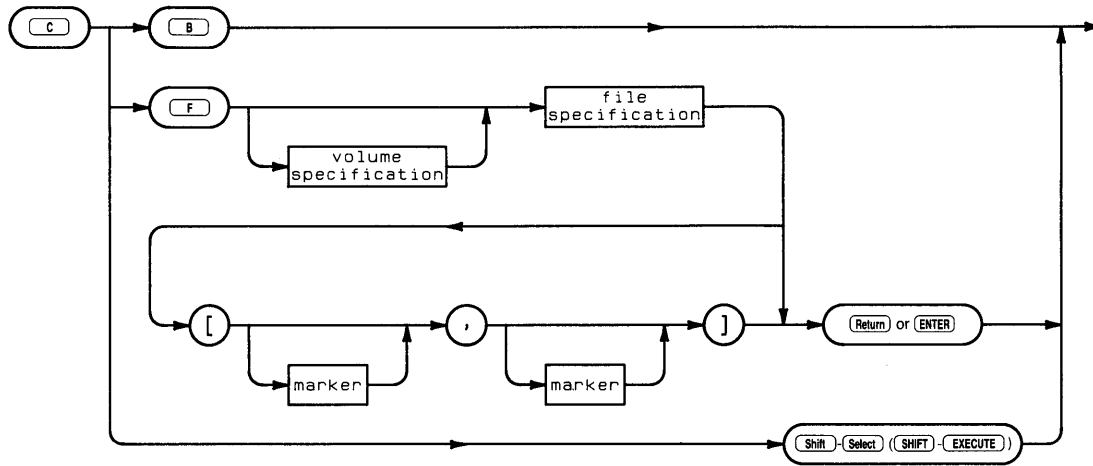
Item	Description	Range
volume name	literal	any valid volume name
file name	literal	any valid file name (do not enter .TEXT suffix)
marker	literal	1 to 8 ASCII characters, excluding CHR(0) thru CHR(31) and CHR(127)

Semantics

The Copy prompt:

```
>Copy: Buffer File <esc>
```

The Copy command provides a way of moving or duplicating text in a file and copying text from another file. These are the Buffer and File options. Pressing ⬚C⬚ (for Copy) and ⬚B⬚ (for Buffer) results in the contents of the copy buffer being inserted into your text at the current cursor position. The screen displays the new text and the Editor prompt.

The copy buffer is filled with the text involved in the most recent Delete, Insert or Zap command. Its contents are cleared with the Margin command. Margin clears the copy buffer regardless of the settings in the environment. Doing a Copy (from a File) also clears the copy buffer. A subsequent Copy from Buffer command generates the message:

```
>ERROR: Invalid copy.  <space> continues.
```

Any subsequent Delete, Insert or Zap refills the buffer (destroying its previous contents), and copying from a file clears the contents of the buffer. However, doing a Copy (from Buffer) does not alter the buffer's contents. Neither do any cursor control movements or commands. Therefore, you can make multiple copies of the same text in different locations by repeatedly positioning the cursor and pressing ⬚C⬚ ⬚B⬚.

4-30 The Editor

To Copy from a file, press ⎡C⎤ ⎡F⎤. The screen displays:

```
>Copy: File [marker,marker] ?
```

The Editor is requesting a file specification and two marker names. The volume name may be omitted if the file in question is on the default volume. The volume (specified or default) must be on-line. Specification of the two previously set markers (see Set command) is optional but, if given, the marker names must be enclosed in square brackets and separated by a comma. Remember, only TEXT type files have markers.

If markers are specified, only the text between those two markers is copied. If no markers are specified, the entire file is copied. Only one marker has to be specified. If it is the first marker (i.e., followed by a comma), the text is copied from the marker position to the end of the specified file. If only the second marker is given (i.e., preceded by a comma), the text is copied from the beginning of the specified file to the position of the marker. The copy occurs at the cursor's position when the Copy command was entered. You can exit the command before all specifications are complete by clearing the line and pressing ⎡Return⎤ or ⎡Enter⎤.

After typing the appropriate information and pressing ⎡Return⎤ or ⎡Enter⎤, the Editor displays:

```
>Copy...
```

This shows that the specified text is being copied into your current text. When the operation is complete, the Editor prompt reappears and the screen displays all or part of the text that was copied.

Delete

Delete removes text from the current file.
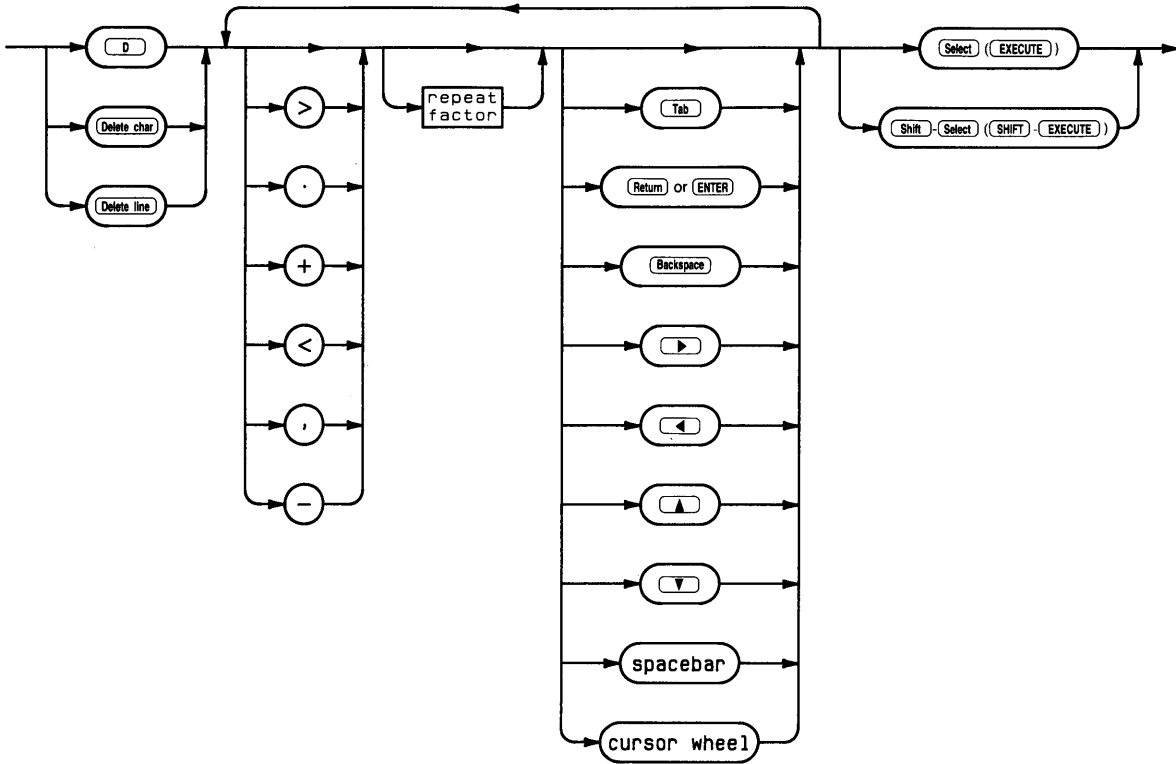
Item	Description	Range
repeat factor	integer numeric constant	1 thru 9999 (1 thru 4095 for Tab)

Semantics

The Delete prompt:

```
>Delete: < > <Moving commands> [<sel> deletes, <esc> aborts]
```

The Delete command enables you to remove text and fills the copy buffer with the deleted text. Delete uses the cursor position when the command is entered as a starting point. Subsequent cursor movement by any cursor control key causes text to be removed between this point and the new cursor position. Text can be recovered by moving the cursor back toward the starting point.

Direction applies in the Delete command and is shown by > (forward) or < (backward) in the Delete prompt. Movement generated with [Tab], [Return] or [Enter] and space bar takes place in the direction shown; forward movement is from the cursor toward the end of the file, and backwards movement is from the cursor toward the beginning of the file. Direction can be changed while in the Delete command by pressing > . or + (for forward) or < , or - (for backward).

Repeat factors are available within the Delete command. For example, pressing [D] (for Delete) and then 9 [Return] or [Enter] will remove 9 lines of text in the current direction, starting at the cursor position.

Delete fills the copy buffer with the deleted text and thus provides a means of moving or duplicating text. See the example in the section "A Sample Editor Session".

To exit the Delete command press [Select] (**EXECUTE**) or [Shift]-[Select] ([Shift]-**EXECUTE**). [Select] confirms the deletion, returns the Editor prompt and displays the cursor at its position when [Select] was pressed. [Shift]-[Select] aborts all changes made since Delete was entered, returns the Editor prompt and displays the cursor at its position when Delete was entered.

Note that the copy buffer is filled by whatever is deleted; whether the command is exited with an [Select] or [Shift]-[Select] makes no difference to the copy buffer.

=

Equals ($\boxed{=}$) positions the cursor at the anchor's location.
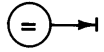
Semantics

The equals sign ($\boxed{=}$) is a cursor positioning command. It moves the cursor to the "anchor" (i.e., the beginning of the most recent item Adjusted, Found, Inserted, or Replaced). The anchor is also used by the Zap command.

Find

Find moves the cursor to an occurrence of a specified string.
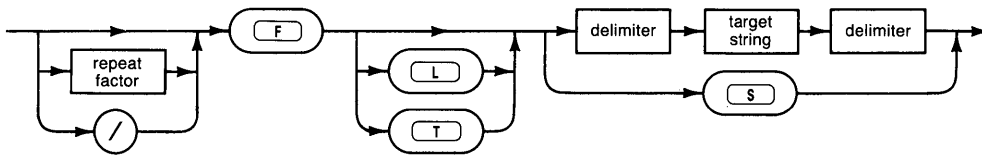
Item	Description	Range
repeat factor	integer numeric constant	1 thru 9999
delimiter	literal	any valid delimiter; must be used in matched pairs
target string	literal	1 thru 128 characters

Semantics

The Find prompt:

```
>Find[1]: L <target>=>
```
or
```
>Find[1]: T <target>=>
```

The prompt displayed depends on whether the "Token" definition in the Editor's environment is set to True or False. If set to True, the first prompt is displayed; if False, the second is shown. These are explained below.

In its simplest form, the Find command is executed by pressing ⌈F⌉ and specifying a string surrounded by delimiters. Upon typing the final delimiter, the cursor is positioned at the end of the first occurrence of the specified string in the current direction, if found. If the pattern is not found, then this message is displayed: `ERROR: Pattern not found. <space> continues`. Pressing the space bar returns you to Edit mode at the previous location in the file.

The Find command moves the cursor and sets the anchor (used by Zap) at the location of the target string. In this context, a "string" is a contiguous series of non-control ASCII characters surrounded by delimiters. Delimiters are separators that signify to the Editor the beginning and end of the string. They can be any non-alphanumeric characters such as / ' . and >.

Don't use a delimiter that appears in your string. Delimiters must be matched pairs; if you use ($) to signify the beginning of a string, you must use ($) to signify the end of the string. The maximum length of a target string is 128 characters.

The Find prompt shows the current direction. When searching for a string occurrence, Find looks for that string between the cursor position when the command was entered and either the end of the file (if direction is forward >) or the beginning of the file (if direction is backward <).

Repeat factors are available with the Find command but must be typed before the Find command is initiated. If a repeat factor is used, the Editor positions the cursor at the end of that occurrence of the string. For example, typing ⟨8⟩⟨F⟩/the/ results in the cursor being positioned at the end of the eighth occurrence of `the`. The slash (/) operates in a similar way but signifies the last occurrence of the specified string in the current direction. If no repeat factor or slash character is specified it defaults to the value 1 and the Editor attempts to find the first string occurrence. The Find prompt displays this value in square brackets.

After pressing ⟨F⟩, the prompt on your screen contains either an L or T for "literal" or "token" modes. Literal and token are mutually exclusive; if one option is shown as available, the other is automatically the default. If L is shown in the prompt and you want to use the token search mode, simply type in the target string surrounded by delimiters. The search will take place in the default mode (in this case, token). To do the same search in the literal mode, press ⟨L⟩ then type in the string as before. The Find command then searches for a literal form of the string.

A literal string is exactly that — a literal string of characters either isolated or embedded in a word or paragraph. **A token string** is one which is isolated by delimiters. Delimiters in this context are any ASCII characters except numbers or alphabetic characters. Blanks, commas, and periods are common delimiters in English text because they separate words.

To illustrate literal and token searches, the following example assumes the direction is forward (>) with the cursor located at or before the start of the sentence shown. In the sentence `That's my hat!`, a token search for `hat` moves the cursor behind the last word `hat` in the sentence whereas a literal search for `hat` moves the cursor behind the `hat` embedded in `That's`.

The "same" option is another feature of the Find command. Same refers to the most recent target string used in either the Find or Replace command. Suppose you typed the sequence ⟨F⟩⟨L⟩ `*galactic*`. After pressing the final delimiter (*), the Editor moves the cursor behind the first literal occurrence of the target string `galactic`. Then typing ⟨F⟩⟨L⟩⟨S⟩ results in the cursor moving behind the next literal occurrence of the same target.

Note

If a Replace has been done since the last Find operation, the target string used by the "same" option is now the target specified in the Replace command.

Searches are sensitive to the case of the characters (upper and lower case) unless Ignore case and Token are set to True in the Environment. Type ⟨S⟩ and ⟨E⟩ to Set the Environment. Type ⟨I⟩ and ⟨T⟩ to set Ignore case to True. Type ⟨T⟩ and ⟨T⟩ if Token is not already True. After these two conditions are met, the Editor treats both the target string and each token string as uppercase.

Find is one of the commands that sets the anchor used in the Zap command and accessed by the Equals command.

The ⟨Shift⟩-⟨Select⟩ (⟨Shift⟩-**EXECUTE**) can be used to abort the Find command before all specifications are complete. ⟨Select⟩ cannot be used with the Find command. The command is executed immediately when the final delimiter (or ⟨S⟩ if "same" is used) is typed.

Insert

Insert opens a window in the current file for the subsequent insertion of text.
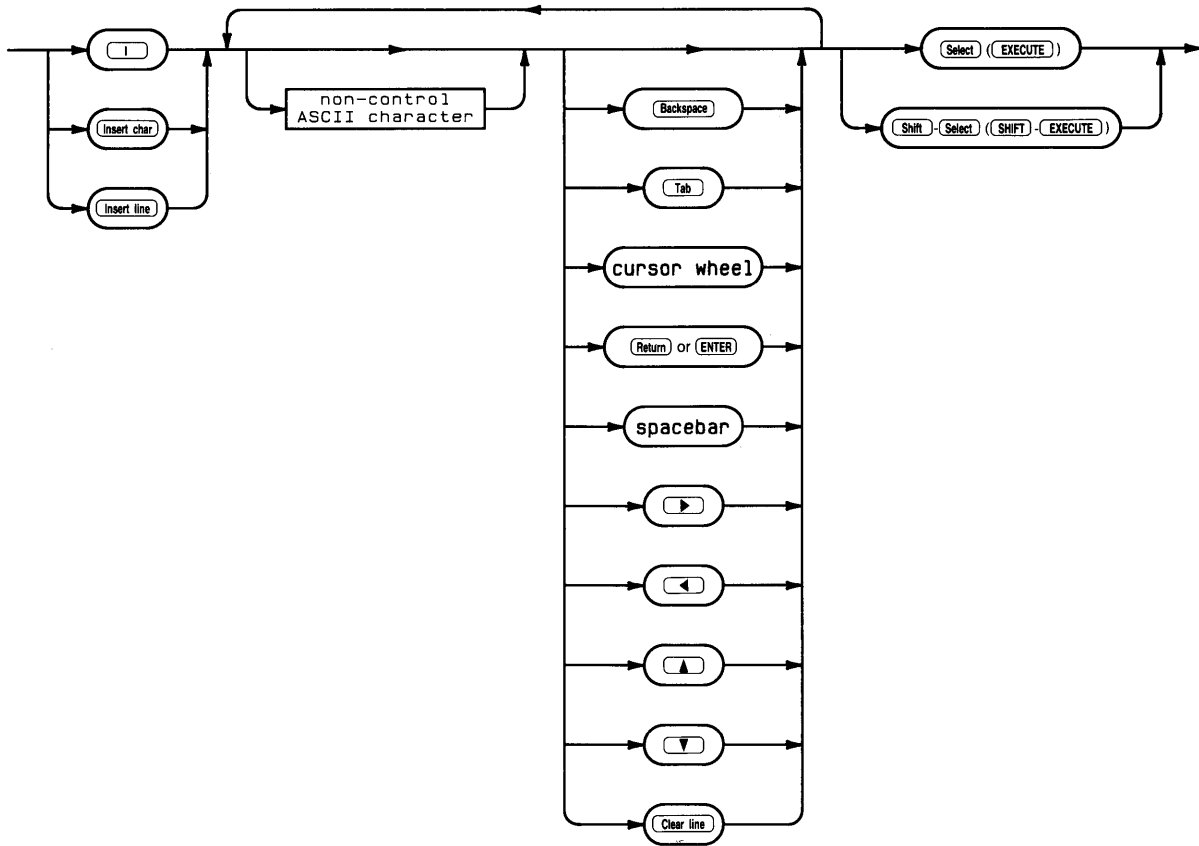
Item	Description	Range
non-control ASCII character	literal	CHR(32) thru CHR(126)

Semantics

The Insert prompt:

```
>Insert: Text <bs>, <clr ln> [<sel> accepts, <esc> escapes]
```

The Insert command opens a window in the text file directly in front of the cursor position for text creation. When initiated by pressing `I` or `Insert char` or `Insert line`, the text from the cursor to the right edge of the screen is shifted to the right. Insertion always takes place directly in front of the cursor location when Insert was entered. Any sequence of non-control ASCII characters may be inserted and any cursor control key may be used. However, unless the movement generated by the cursor control keys is backward, they produce question marks (?) in the text. You can `Back space` to delete a character or press `Clear line` (`Clear line`) to delete the most recently inserted line. `Clear line` is available only after a line of text has been inserted. Backspacing past the point at which Insert was entered is not possible.

The way in which text insertion takes place depends on flags or parameters set in the Editor's environment. These flags have default values supplied by the Editor but can be changed with the Set command. The ones that concern you here are Filling and Auto indent. These two options generally have opposite values. Most of what you need to know about Filling and Auto indent can be summed up in one sentence: If you are writing program source text, set Filling to false and Auto indent to True (Program mode); if writing regular text, set Filling to True and Auto indent to False (Document mode).

Filling, when set True, performs both "wrap around" and "margining" functions. As inserted text approaches the Right margin (another environment option), the Editor attempts to fit the words on the current line. If a word would cause the line to extend beyond the right margin, it is automatically shifted to the next line (i.e., the system supplies a `Return`). When the insertion is completed by pressing `Select` (**EXECUTE**), all text following the cursor in the current paragraph is margined. Margining adjusts the text to fit between the environment's margins and also compresses blanks in the text. You can have two blanks following these four characters: ? . : !. All other blanks are compressed into a single blank character.

Note that the Editor's definition of a paragraph is **ANY** text delimited by any combination of blank lines, lines having the Command character as the first non-blank character in a line, or the beginning and end of a text file. The Command character is yet another of the environment's options; see the Set command for more details.

Note

As the definition of a paragraph infers, the Editor does not distinguish tables from other kinds of text material. Any insertions within a table will result in the table being margined (i.e., collapsed) if Filling is set to True, autoindent to false, and the insertion is exited with `Select` (**EXECUTE**). Use the Set command to set Filling to False before inserting in a table or list. (`Shift`-`Select`) will NOT restore the text to its original state).

If Filling is False, a beep is generated as you approach the end of the line, signaling you to press [Return] or [Enter] the same way a bell on a typewriter does. If you continue to insert text past the last visible column on your screen, the Editor accepts the characters and shows you that they are outside of the display area by placing an exclamation point (!) in the final column. To access these characters, complete the insertion by pressing [Select] (**EXECUTE**), position the cursor before the last visible word on the line and press [I] followed by [Return] or [Enter] to insert a carriage-return (and thus break the line at that point).

If Auto indent is True, pressing [Return] or [Enter] automatically places the cursor in the same starting column as the previous line. When Auto indent is False, the cursor is positioned according to either the Left or Paragraph margin in the environment.

If Insert is confirmed with [Select] (**EXECUTE**) and Filling is True and Auto indent False, all text following the insertion in the same paragraph is margined according to the Right, Left, and Paragraph margin values in the environment. Also, the entire insertion is stored in the copy buffer so you can copy the same text elsewhere if you wish. If Insert is aborted with [Shift]-[Select], regardless of the options set, all changes are aborted, the copy buffer is cleared, and the text and cursor appear as they did when the command was entered.

The Insert command sets the **anchor** (used by the Zap command) at the position where Insert was initiated. The anchor is set regardless of whether [Select] or [Shift]-[Select] is used to exit the command.

Jump

The Jump command repositions the cursor.
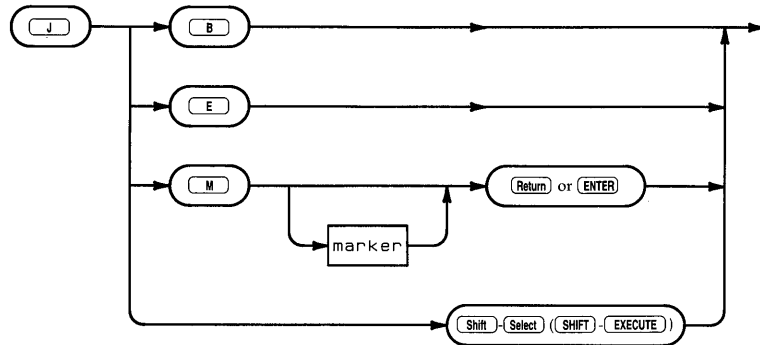
Item	Description	Range
marker	literal	1 to 8 ASCII characters excluding: CHR(0) thru CHR(31), and CHR(127)

Semantics

The Jump prompt:

```
>JUMP: Begin End Marker <esc>
```

The Jump command moves the cursor to the beginning or end of a text file or to a previously defined marker. The command has no other effects; it merely repositions the cursor. To Jump to the beginning of your file, press ⬚J ⬚B. To Jump to the end of your file, press ⬚J ⬚E.

You can also Jump to a marker by pressing ⬚J ⬚M and typing the name of any previously set marker in the file followed by ⬚Return or ⬚Enter. Marker names are defined with the Set command. A legal marker name is any sequence of up to eight non-control ASCII characters (control characters are deleted by the system). They can actually be longer than this, but the Editor only pays attention to the first 8 and truncates the rest.

Also, marker names are not case-sensitive. The Editor converts all marker names to uppercase letters so they can be typed using any desired combination of uppercase and lowercase letters. There is a 10 marker limit per text file. See the Set command for more information on markers.

Margin

Margin formats all text in the current paragraph to fit the margins set in the environment.

(M)▶◀

Semantics

Margin is disabled and the system generates an error message unless the environment's Auto indent is False and Filling is True when the command is executed.

The Margin command provides a means of formatting paragraphs in your file. A paragraph is defined by the Editor to be **ANY** text delimited by any combination of blank lines, lines having the Command character as the first non-blank character in a line, or the beginning or end of a text file. See the Set command for details on the Command character.

Upon initiating Margin (by pressing ⌈M⌉), the Editor takes all the text in the current paragraph (the one where the cursor is) and forces it to fit within the Left, Right, and Paragraph margin boundaries of the environment. After margining, the first line of the paragraph begins at the column specified by the Paragraph margin setting and the rest of the text conforms to the Left and Right margin settings. If a word would exceed the Right margin it is "wrapped around" to the next line.

Two blanks are allowed following the four characters: ? . : !. All other blanks are compressed into a single blank character.

Since the Command character in the environment delimits a paragraph, you may want to use it as the first character in each line of tables or lists which you do not want margined. See the Set command for more information on the Command character.

Note

If a table or list fits the definition of a paragraph, the Margin command will definitely reformat that text. Exiting the Insert command with ⌈Select⌉ (**EXECUTE**) also uses some of the Margin routine so be aware that these commands can potentially "collapse" a table or list.

The Margin command has no parameters and its effects cannot be aborted. When writing program text or tables, it is advised that Auto indent be set True, Filling be set False and the Paragraph margin be equal to the Left margin.

Note

The Margin command clears the contents of the copy buffer regardless of the settings of the Auto indent and Filling options.

Page

Page moves the cursor one or more pages (screenfuls) in the current direction.
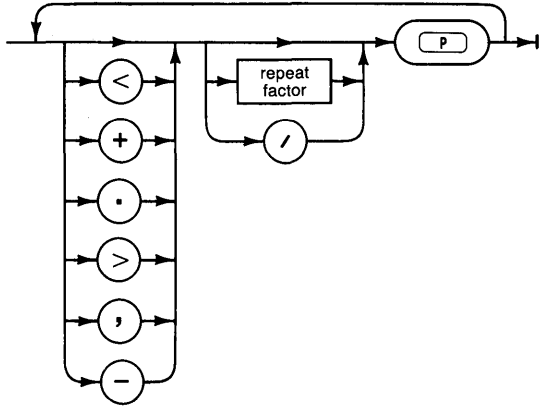
Item	Description	Range
repeat factor	integer numeric constant	1 thru 1000

Semantics

The Page command lets you move rapidly through a text file by repositioning the cursor one or more pages forward (>) or backward (<) in a file. Page is executed by pressing ⟨P⟩, and its movement occurs relative to the position of the cursor. Page moves the cursor in the direction displayed by the Editor prompt when the command is entered. The direction can be changed by pressing >

Repeat factors are available in the Page command. For example, to move the cursor 3 "screens" or pages in the file, press ⟨3⟩ ⟨P⟩. The slash character (/) can be used in place of an integer repeat factor. Pressing ⟨/⟩ ⟨P⟩ results in the cursor moving to the end of the file (if direction is >) or the beginning of the file (if direction is <). If neither repeat factor nor slash is specified, the default is 1 and the cursor moves one page in the current direction.

⟨Select⟩ (**EXECUTE**) and ⟨Shift⟩-⟨Select⟩ are not available in the Page command. The command is immediately executed when ⟨P⟩ is pressed.

Quit

Quit leaves the Editor with various exit options.
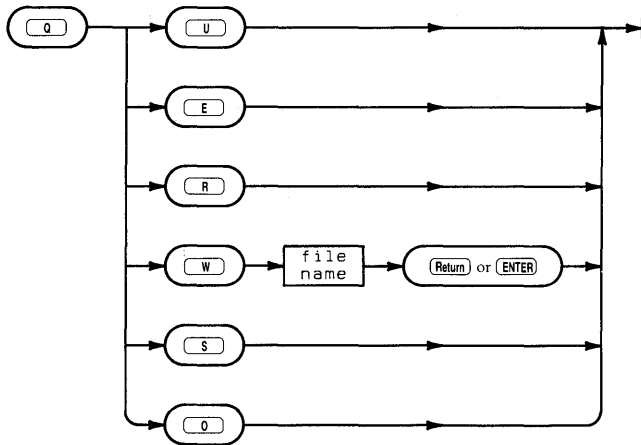
Semantics

The Quit prompt:

```
>Quit:
    Update the workfile and leave
    Exit without updating
    Return to the editor without updating
    Write to a file name and return
    Save as file new file DOC:BINSEARCH.TEXT
    Overwrite as file DOC:BINSEARCH.TEXT
```

The Quit command allows you to exit the Editor and store your file on mass storage. The last two quit options shown above are only available if the file existed before the editing session.

Quit is initiated by pressing ⌈Q⌉ from the Editor's prompt. Choose any of the options displayed by pressing the first letter of the option.

Pressing ⌈U⌉ (for Update) results in the contents of the text in the computer's memory being written to a text file on the system volume under the name WORK.TEXT. This workfile may or may not be associated with another file name (see the Get and Save commands in the Filer chapter). After writing the file, the system reports the file's size (in number of bytes and blocks) and displays the Main Command Level prompt.

Pressing ⌈E⌉ (for Exit) either immediately exits to the Main Command Level or displays:

```
  Are you sure you want to exit without updating?
      Type Yes  to Exit without update
      Type No   to Return to Editor
```

This message is displayed only if changes have been made to the text file in the current editing session. If no changes have been made, the system immediately goes to the Main Command Level when ⌈E⌉ is pressed. It also exits to the Main Command Level if you respond by pressing ⌈Y⌉. Responding with ⌈N⌉ returns you to the Editor.

Pressing ⬚R (for Return) returns you to the Editor with the cursor located where it was when Quit was entered.

Pressing ⬚W (for **Write**) causes the system to prompt you for a file name. If a volume ID is not given, the default volume is used. The volume PRINTER: may be specified. This results in the file being listed to the system printer.

If you use the Write option and the file already exists, the Editor displays this prompt:

```
>Quit:
FILE.TEXT exists ...
   Rewrite then purge old
   Overwrite
   Purge old then rewrite
   None of the above
```

Rewrite then purge old is like the Save command. An attempt is made to write the new file before purging the old.

Overwrite removes the original file and then attempts to write the new version in its place. On SRM units, or for HFS discs, duplicate links, and passwords for SRM, will be preserved. On a local disc, the file may not fit if the new version is larger than the old.

Purge old then rewrite removes the original file and then attempts to write the new file in the biggest space on the disc. This alternative gives you the best chance that there will be room for the new file.

If the write fails, whether you "Overwrite" or "Purge old then rewrite", the original copy of the file is gone and the only copy of the file is in the Editor's memory. It is advisable to save it on another disc as soon as possible.

None of the above returns you to the Editor. You may Quit again and write the file with a different name.

Pressing ⬚S (for **Save**) results in the file being written to the original volume and file.

If you try to Save a file and you get the message:

```
>ERROR: No room on vol <space> continues.
```

Press the spacebar to continue. You could put in another disc with enough space, then Quit and Write it on the new disc. Alternatively, you can Quit and Overwrite the file.

Pressing ⬚O (for Overwrite) is designed for SRM systems and HFS discs. The Overwrite option allows all duplicate links (and passwords for SRM) to remain intact. On a local non-HFS disc, Overwrite may not work if the file has been enlarged. If this happens, press the spacebar to continue, Quit and Save again. The previous Overwrite removed the original file. Now the Save will try to save the file in the largest space on the disc. If this does not work, you must put the file on another disc.

Replace

Replace does one or more substitutions of a specified string for another string.
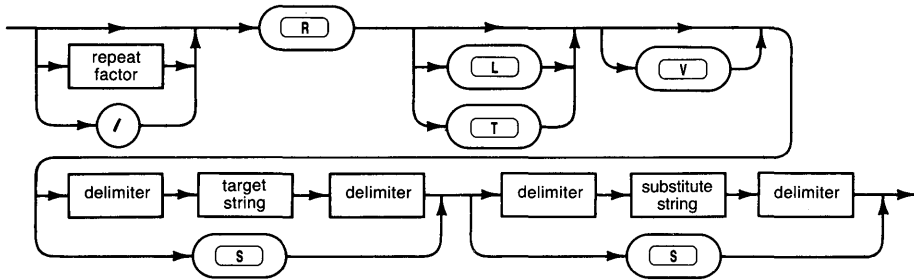
Item	Description	Range
repeat factor	integer numeric constant	1 thru 9999
delimiter	literal	any valid delimiter; must be used in matched pairs
target string	literal	1 thru 128 characters
substitute string	literal	1 thru 128 characters

Semantics

The Replace prompt:

```
    >Repl[1]: L  V  <targ><sub>=>
or  >Repl[1]: T  V  <targ><sub>=>
```

The prompt displayed depends on whether the "Token" definition in the Editor's environment is set to True or False. If True, the first prompt is displayed; if False, the second is shown. The Token definitions are explained below.

The Replace command allows you to substitute one string for another in your text file. The anchor (used by the Zap command and accessed by the Equals command) is set at the location of the replacement. Replacements can be done to a single, all, or only certain occurrences of a string.

In its simplest form, the Replace command is executed by pressing ⌈R⌉ and specifying two strings — a target and substitute — each surrounded by delimiters. The target and substitute strings may be different sizes. Upon typing the final delimiter, the first occurrence of the target string is replaced by the substitute string and the cursor is positioned at the end of the substitution.

A target string (the one that you want replaced) must be supplied. A string is a contiguous series of non-control ASCII characters surrounded by delimiters. Delimiters signify the beginning and end of a string and are characters such as: / ' . and >.

A substitute string (what you want the target string changed to) must also be supplied with delimiters. The substitute may be an empty (null) string.

Don't use a delimiter that appears within your string. Delimiters must be matched pairs; if you use $ to signify the beginning of a string, you must use $ to signify its end. The substitute string can have a different set of delimiters than the target string and the two strings may be of different sizes. The maximum length of either string is 128 characters.

After pressing ⌈R⌉, the prompt on your screen contains either an ⌈L⌉ or ⌈T⌉ for "literal" or "token" modes. Literal and token are mutually exclusive; if one option is shown as available, the other is automatically the default. If ⌈L⌉ is shown in the prompt and you want to use the token search mode, then type in the two strings and their delimiters. The replacement takes place in the default mode, in this case, token. To do the same replacement in the literal mode, press ⌈L⌉ and type in both strings as before. The Replace command then searches for a literal form of the string.

A literal string is exactly that — a literal string of characters either isolated or imbedded in a word. **A token string** is usually a word (a string isolated by delimiters). Delimiters, in this context, are any ASCII characters except numbers or alphabetic characters — they do not have to be matched pairs. Blanks, commas, and periods are the most common delimiters in English text because they separate words.

To illustrate literal and token replacements, the following example assumes the direction is forward (>) with the cursor located at or before the start of the sample sentence. In the sentence `That's my hat!`, a token replacement for `hat` with `umbrella` replaces the last word `hat` in the sentence with `umbrella` whereas a literal replacement would substitute `umbrella` for the `hat` imbedded in `That's` (resulting in `Tumbrella's my hat!`).

Direction applies in the Replace command and is shown by the first character in the command's prompt. If the direction is forward (>), the replacement occurs between the cursor position and the end of the file; if backward (<), between the cursor and the beginning of the file.

Repeat factors are available for the Replace command but must be typed before the command is initiated (before ⌈R⌉ is pressed). A repeat factor causes that number of substitutions to be made. If not specified, the repeat factor defaults to 1. A slash character (/) may also be used to change all occurrences of the specified string in the current direction. The repeat factor (or slash character) is displayed in brackets [ ] in the command's prompt. The repeat factor works differently when the Verify option is used.

The Verify option lets you choose whether or not to make a particular replacement. The combination of a repeat factor with Verify allows you to replace only certain occurrences of a string in the file. For example, after pressing ⌈2⌉ ⌈R⌉ ⌈V⌉ and typing in the target and substitute strings, the Editor moves the cursor to the first occurrence of the target string and prompts:

```
>Rpl[2]: <sh-exc> aborts,R replaces,' ' doesn't
```

To confirm the replacement, press ⌈R⌉. To skip to the next replacement (if any), press the space bar. While using Verify, pressing ⌈Shift⌋-⌈Select⌉ (⌈Shift⌋-**EXECUTE**) aborts the operation but retains all replacements made up to that time.

The "same" option is available with Replace and refers to either the most recent target string (used in a Find or Replace) or the most recent substitute string (used only in Replace). Which string it signifies (target or substitute) depends on where it is used in the Replace command. To use "Same", simply press ⎡S⎤ in place of the delimited string. If you type ⎡S⎤ followed by a delimited string, the most recent target is replaced with the specified string. If you type a delimited string followed by ⎡S⎤, the specified target is replaced with the last substitute. Both strings may be specified by typing ⎡S⎤ ⎡S⎤. The current assignments of the "same" patterns can be seen by pressing ⎡S⎤ ⎡E⎤ (see the Set command for more details).

Note

If a Find has been done since the last Replace, the target string used by the "same" option is now the target specified in the Find command.

The "Ignore case" option applies to the Replace command. Type ⎡S⎤ and ⎡E⎤ to Set the Environment. Type ⎡I⎤ and ⎡T⎤ to set Ignore case to True. Type ⎡T⎤ and ⎡T⎤ if Token is not already True. The target string and all **token** strings in the text are treated as upper case. When a match is found, the token string is replaced with the substitute string. The case of the substitute string is not affected by the Ignore case option.

The Replace command can be aborted before all specifications are complete by pressing ⎡Shift⎤- ⎡Select⎤ (⎡Shift⎤-**EXECUTE**). (Subsequent use of the "same" option after aborting the Replace command may give you unwelcome results).

Set

Set defines markers and alters the environment in which your text operations occur.
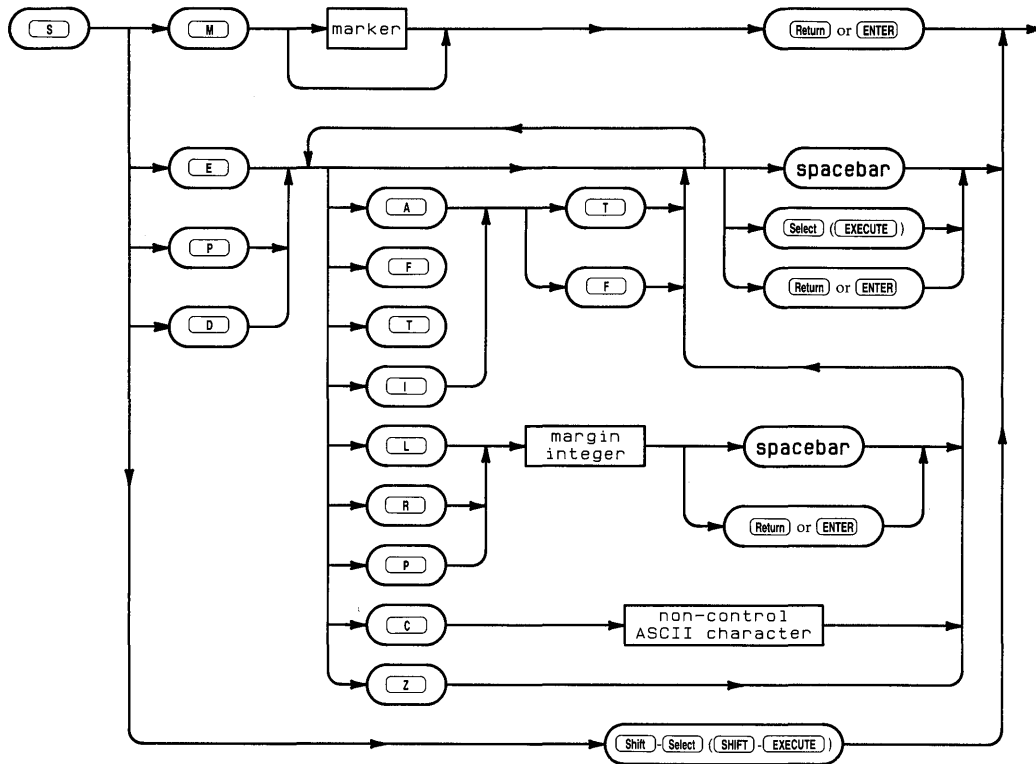
Item	Description/Default	Range Restrictions	Recommended Range
marker	literal	1 to 8 ASCII characters, CHR(32) thru CHR(126)	—
margin integer	integer numeric constant	0 thru 9999; left margin must be less than right margin	0 thru 49 for 50-column displays; 0 thru 79 for 80-column displays
non-control ASCII character	literal	CHR(32) thru CHR(126)	—

Semantics

The Set prompt:

```
>Set: Env Mrk Prog Doc <esc>
```

Set lets you define markers and various environment parameters. Markers are Set by moving the cursor to where you want the marker, pressing ⎡S⎤ ⎡M⎤ (for Set Marker) and typing in a marker name followed by ⎡Enter⎤. A marker name is any sequence of up to eight non-control ASCII characters. The Editor accepts more than eight characters but truncates anything longer. All non-printing characters (those with an ASCII value of either 127 or in the range of 0 to 31) are deleted by the system. The Editor converts these names to uppercase so they can be typed in whatever form is convenient.

No more than ten markers can be set in a file. If you attempt to set more than ten, the Editor displays the markers in a numbered list and prompts you for the number of the marker you wish to replace. All markers can be removed by giving the Zap marker command. Markers are used with the Jump and Copy commands and their names are shown in the environment display. The locations of the markers are not shown so the use of meaningful marker names is advised.

Pressing ⎡S⎤ ⎡E⎤ (for Set Environment) displays the current environment and allows you to change the environment's parameters. When entering the Editor with a new file, the default environment is the Program environment which looks like:

```
>Environment: {options} <sel> or <sp> leaves
     Auto indent   True
     Filling       False
     Left margin   0
     Right margin  78
     Para margin   5
     Command ch    ^
     Token def     True
     Ignore case   False
     Zap markers
     275 bytes used, 348909 available.

     Patterns:
       <target>= 'inery', <subst>=  'inary'

     Markers:
         TOP           FIX

     File BINSEARCH.TEXT
     Date Created: 10-11-82   Last Used: 10-11-82
```

Patterns and Markers are only shown if they have been set. The heading near the bottom displays a file name if the Editor is entered with a specified file. Whenever a TEXT file is saved on mass storage, the current environment is saved with it and becomes the default environment when that file is used by the Editor. (Note that the environment is not saved with Data, ASCII, or HP-UX compatible files.)

The environment also displays how many bytes of memory have been used and how many are still available for use in the Editor. The total number of bytes (used and available) depends on the amount of memory in your machine.

To change a parameter in the environment, press the first letter in the parameter's name. The cursor is automatically positioned at the item to be changed and the new value must be typed. If the parameter needs a number (as in Left, Right, and Paragraph margins), then the number must be followed by pressing ⌈Return⌉ or ⌈Enter⌉ or the spacebar. All other parameters accept a single character and return the cursor to the environment's prompt as soon as the character key is pressed.

Automatic indenting is a boolean (with either a true or false value) which affects the Insert and Margin commands. When inserting text with this item set true, pressing ⌈Return⌉ or ⌈Enter⌉ automatically moves the cursor to the next line at the same starting column as the previous line. This indenting feature is useful when writing Pascal programs so it is set true for the Program (default) environment.

When Auto indent is true the Margin command is disabled. When Auto indent is false, pressing ⌈Enter⌉ places the cursor on the next line at either the Left margin or Paragraph margin (as currently defined in the environment).

Filling is another boolean value which affects the Insert and Margin commands. It usually has a value opposite that of Auto indent. When set True, filling causes automatic "wrap around" of text. If a word is too long to fit on the current line (as defined by the Right margin value), it is carried or wrapped around to the next line and no carriage-return (⌈Return⌉ or ⌈Enter⌉) is necessary. Another effect of this parameter being set true (and auto-indent False) is that an Insert completed by pressing ⌈Select⌉ (**EXECUTE**) causes all text following the insertion in that paragraph to be margined or filled according to the current values of the Left, Right and Paragraph margin settings. All blanks in the text are then compressed to a single blank (though two blanks are allowed following the characters: ? . ! :). The Margin command only works when Filling is set True and Auto indent is set False.

With Filling set False, the wrap around and margining functions are disabled. When approaching the end of a line, the system generates a "beep" to inform you that you need to press ⌈Return⌉ or ⌈Enter⌉ to go to the next line. If you type past the display area of the screen, an exclamation point (!) is shown in the last column. The text, though not visible, is maintained in the computer's memory.

The Left margin may be set to any integer between 0 and 9999. Numbers longer than 4 digits are truncated by the system. The Left margin must be less than the Right margin setting or an error message is generated when you attempt to exit the environment.

The Right margin setting has the same numerical limitations as the Left margin. Unless you have a particular reason for doing so (like making full use of a 132-column printer), it is not a good idea to set this margin beyond the right column display limits of your screen because the text will not be visible. A line length limitation does exist for the compiler which will only process up to the first 110 characters on a line of a program source.

The Paragraph margin can be set to any positive integer up to 4 digits. This setting determines the indention that the first line in each "paragraph" will get. This occurs when Filling is set false (while inserting text) or when Margin is used. Note that a paragraph as defined by the Editor is any text surrounded by blank lines or by lines beginning with the Command character (discussed below). The beginning and end of a file will also delimit paragraphs.

The Command character can be any non-control ASCII character. If this character is the first non-blank character in a line, the Margin command treats the line as if it were blank. The line is not margined and it is considered to be the beginning or the end of a paragraph. The default Command character is the (^) character.

Token is a boolean used by the Find and Replace commands. When Token is set True, the default value for Find or Replace becomes token and the command's prompt displays the literal option. (Token and literal refer to the type of target string searches that take place in these commands). Conversely, if Token is False, the default value for Find and Replace is literal and the command's prompt displays token as an option.

The **Ignore case** command affects searches in the Find and Replace commands. If "Ignore case" is left as False, then "string" and "STRING" and "String" are not treated as equal. If "Ignore case" is set to True, they are treated as equal. This only works when the Environment's **Token** mode is True or if you type a "T" before typing the target string.

The **Zap markers** command removes all markers from the file.

The environment display is left and the Editor's main prompt returned by pressing ⌈Return⌉ or ⌈Enter⌉, ⌈Select⌉ (**EXECUTE**) or the spacebar. The current environment settings are only saved in the mass storage file if it is of type TEXT.

Although there is only a single environment associated with a text file, the environment may be set to one of two predefined configurations: the **Program environment** (by pressing ⌈S⌉⌈P⌉) and the **Document environment** (by pressing ⌈S⌉ ⌈D⌉). These configurations optimize the various environment parameters for writing programs or regular (non-program and non-tabular) text, respectively. When either predefined environment is Set, the current environment is displayed and any of its parameters can be changed. If you want to change just one or two parameters, use ⌈S⌉ ⌈E⌉ to get into the existing environment.

Changes made to the environment cannot be aborted but the parameters may be changed as many times as desired.

Verify

Verify refreshes the screen display from memory.

Semantics

The Verify command has no options; it is executed immediately by pressing $\boxed{\text{V}}$. Verify causes the Editor to refresh or update the current screen display from memory, move the current line (the one where the cursor is) to the middle of the screen, and display the Editor prompt. If the cursor is located in the first screenful of text (23 to 47 lines depending on the display) when Verify is used, the line containing the cursor is not moved.

eXchange

eXchange replaces text character for character at the cursor position.
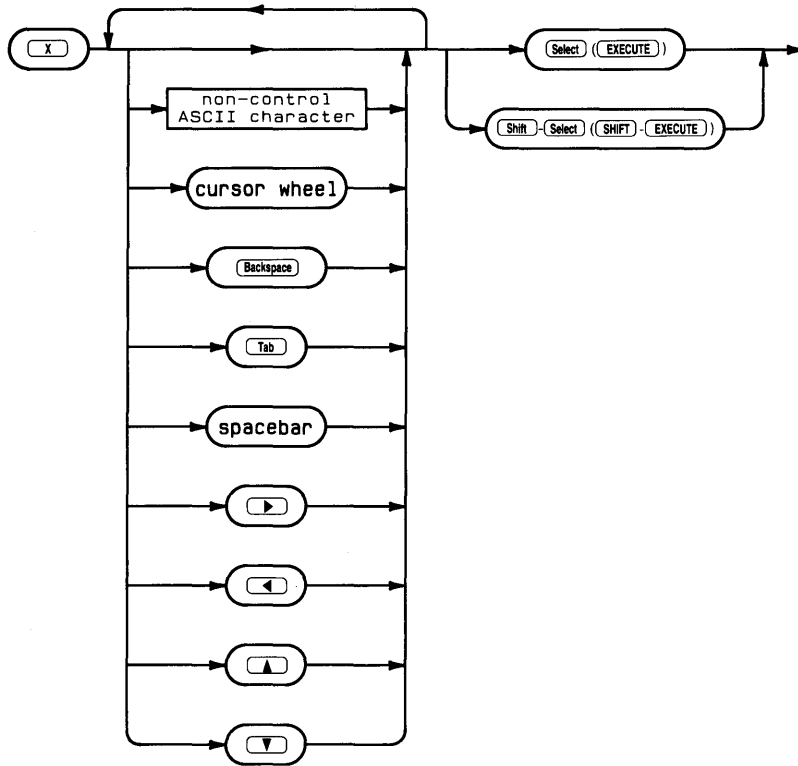
Item	Description	Range
non-control ASCII character	literal	any valid ASCII character, excluding CHR(0) thru CHR(31), and CHR(127)

Semantics

The eXchange prompt:

```
Xchnge: Text <bs> <esc> aborts <sel> accepts
```

The eXchange command lets you exchange text character for character at the cursor position. eXchange operates only on the current line (i.e., the line where the cursor is located when the command is entered). The [Back space], [◄] and cursor wheel in backspace direction move the cursor one space back and display the character that was replaced. The [►] and cursor wheel in forward space direction move the cursor forward, up to the end of the line, without changing anything.

Any ASCII character can be used in eXchange, however, use of control characters is not advised. Carriage returns cannot be entered since the command is unable to cross line boundaries. Direction and repeat factors do not apply to the eXchange command. Any cursor control key that does not effectively "backspace" or "forwardspace" the cursor will generate question marks while executing eXchange. Backspacing past the point at which eXchange was entered is not allowed.

eXchange is initiated by pressing [X] and is exited by pressing [Select] (**EXECUTE**) or [Shift]-[Select]. [Select] confirms the exchanges, returns the Editor prompt, and displays the cursor at its position when [Select] was pressed. [Shift]-[Select] returns the copy of the text file in the computer's memory to its state before eXchange was entered, displays the Editor prompt, and shows the cursor at its position when eXchange was entered.

Zap

Zap deletes text and fills the copy buffer with the deleted text.
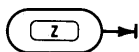
Semantics

The Zap command has no options; it is executed immediately by pressing ⌈z⌋. Zap deletes all text between the "anchor" and the current cursor position and stores it in the copy buffer. The anchor is located at the position in the text where the most recent Adjust, Find, Insert or Replace command was executed. You can confirm the position of the anchor by pressing ⌈=⌋ (the Equals command), which moves the cursor to the anchor.

If more than 80 characters are going to be Zapped, the Editor displays a prompt asking if you wish to Zap anyway. Also, if the Copy buffer is not large enough to store the deletion, a prompt asks if you wish to go ahead and Zap the text. (Use the Set environment command to see how much memory is available; the copy buffer shares this memory with that used to hold the text file in memory).

Recovery of the deleted text is achieved with the Copy (from buffer) command. Zap can also be used to move large chunks of text from one location to another within a file.

Note that the effects of Zap can be surprising since the anchor position is set by four different and commonly used commands (listed above). Therefore, it is a good practice to check the location of the anchor (using the Equals command) before executing a Zap.

The Filer

5

Introduction

This chapter documents the use of the Workstation Filer subsystem. The Filer lets you manipulate files in various ways including moving, listing, duplicating, creating and deleting files. The Filer can handle files on devices with a variety of directory structures and physical characteristics.

Before you read this chapter, you should read "The File System" chapter which defines basic concepts such as files, volumes, and directory organizations.

There are four main sections in this chapter.

- The first two demonstrate how to enter and use the Filer by leading you through a sample Filer session which uses the more common Filer commands.

- The next section, "A Closer Look", presents detailed information about the Filer and its operation.

- The "Filer Commands" section contains an overview or summary of all the Filer commands (useful for quick reference once you are familiar with the Filer) and a semantic and syntactic description of each Filer command, presented in alphabetical order.

Any questions you have about commands covered in the sample session should be answered in the commands section.

Entering the Filer

If your system is not already "up and running", refer to the *Pascal User's Guide* for information on loading the Pascal System. The following prompt must appear on the top line of your screen before you can enter the Filer:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?
```

The prompt tells you that you are at the system's Main Command Level — the level from which all the Pascal subsystems (Compiler, Editor, Filer, etc.) are entered. Entry is accomplished by typing the first character of the subsystem you wish to enter.

Insert the disc labeled **ACCESS**: and press the ⌐F⌐ key. You can use either uppercase or lowercase letters when typing commands at the Main Command Level. However, letter case is important when typing file names. The screen displays:

```
Loading 'ACCESS:FILER'
```

You can use the Permanent command (from the Main Command Level) to keep the FILER code file in memory if you wish. This will allow faster access to the Filer but uses more memory. Chapter 1 explains how to "permanently load" the Filer.

The Filer Prompt

The screen clears and displays the Filer prompt on the top line:

```
Filer: Change Get Ldir New Quit Remove Save Translate Vols What Access Udir ?
```

You are now in the Pascal Filer subsystem. The Filer prompt shows the most common commands used in the Filer and "prompts" you to give the subsystem a command.

The prompt shows only a partial list of the available commands; to see the others, type ⌐?⌐. The prompt line shows the Filer's alternate prompt:

```
Filer: Hfs Bad-secs Ext-dir Krunch Make Prefix Filecopy Duplicate Zero ? [3.2]
```

The alternate prompt displays the revision number of the Filer in brackets. Type ⌐?⌐ again and the main Filer prompt reappears.

All Filer commands are initiated by typing a single key corresponding to the uppercase character of the command shown in the Filer prompt. This is normally the first character of the command name. Uppercase and lowercase command characters are treated as equivalent by the Filer, so the keys may be typed in whatever form is convenient.

Filer operations can be aborted by typing ⌐Shift⌐-⌐Select⌐ (⌐SHIFT⌐-⌐EXECUTE⌐) when a single character is expected and ⌐Shift⌐-⌐Select⌐ followed by ⌐Return⌐ or ⌐Enter⌐ in place of a file specification.

Filer Operations

All of the commands in the Filer operate in one of two ways: the Filer either performs the operation immediately (when you press the letter key for that command) or it requests the information it needs to perform the operation and then does it. The request is generally for a volume specification or a file specification since all of the Filer's commands (except Quit) operate on volumes, directories, and files.

A volume specification identifies a particular volume. This can be done by supplying any of the following: the name of the volume, its associated unit number, a colon (:) to specify the default volume, or an asterisk (*) to specify the system volume. A file specification consists of both a volume specification and a file name; it completely identifies a particular file. All file specifications include a volume specification even if by default. If the volume specification is omitted and only the file name is given, the Filer looks for a file of that name on the default volume.

A Sample Filer Session

Work through the following examples on your machine as you read through this section. Interacting with the computer will teach you more about the Filer than reading the material.

Finding Out What Devices are Accessible

Now that you have the Filer prompt on the screen, press ⌐V⌐. This initiates the Volumes command and the screen now displays the volumes or I/O units associated with the Pascal System. Here is a typical display (yours may vary slightly):

```
Volumes on-line:
   1    CONSOLE:
   2    SYSTERM:
   3 #  MYVOL:
   4 #  ACCESS:
   5 #  SRM_WORK:
   6    PRINTER:
  45 *  SYSTEMO4:
Prefix is - MYVOL:
```

For each volume currently on-line, the display shows the logical unit number and the associated volume name. Volumes #5 and #45 are SRM volumes which you may or may not have.

The "#" beside units 3, 4 and 5 indicates that these are blocked devices. These are used for mass storage. The "*" beside unit 45 indicates that this is the system volume, which is also a blocked device. The system volume is used by the system during certain operations and should be left on-line at all times if possible. **Prefix is -** indicates which is the default volume. The default volume is assumed when no volume identifier is given. The default volume can be changed using the Filer's Prefix command or the Main Command Level's What command.

The Default and System Volumes

When booting, the system designates the mass storage device, from which the system files are read, as the "system volume". The system volume is denoted by an asterisk (*) in the Volumes command display and remains fixed unless the New system volume command or the What command is used at the Main Command Level, or a version of the TABLE program is executed which resets the system volume to a different location, e.g. #45 for SRM.

The Prefix command, because it defines the default volume, lets you specify a particular volume where the Pascal System will look for files when you haven't given a volume name or logical unit number. This is handy in the Filer as well as in other subsystems such as the Compiler or Editor.

The default volume can be indicated with the colon (:) character. For example, to list the directory of the default volume, press ⬜L⬜ (for the List directory command) and answer the prompt by typing : . The Filer then displays the directory of the current default volume.

Changing the Default Volume

You can use the Volumes command (from the Main Command Level) to see what volume is the current default volume. It is listed under the heading **Prefix** in the Volumes display. You can also see what the current default volume is by pressing ⬜P⬜ for the Prefix command. The screen displays:

```
Prefix to what directory ?
```

Respond by pressing ⬜Return⬜ or ⬜Enter⬜. The screen now displays the current default volume. Now press ⬜P⬜ again and in response to the prompt, type **MOJO:**. The screen now displays:

```
Prefix is MOJO:
```

Now, whenever you want to specify a file or group of files on the MOJO: volume, you can just type the file name(s) and the Filer will assume that the file or files specified are on the volume MOJO:.

It is possible to set the default prefix to a flexible disc drive, regardless of the volume inside. This is done by typing:

#3: ⬜Return⬜ or ⬜Enter⬜

while the drive door is open or the drive is empty.

The prefix command is used to set up a working directory on SRM (Shared Resource Management) and HFS (Hierarchical File System) discs as well. If you have an SRM or HFS file named:

#5:/USERS/JOE/PROJECT1/PROGRAMS/FILE

Initiate the Prefix command as usual and specify:

```
#5:/USERS/JOE/PROJECT1/PROGRAMS
```

This sets the SRM or HFS volume as the default volume and /USERS/JOE/PROJECT1/PROGRAMS as the working directory. If the previous working directory had been PROJECT1, then only PROGRAMS need be typed. Now you can specify the file with:

```
FILE
```

If you were to use the Prefix or Udir command again to set the default prefix to another volume (not on the same unit), the working directory and volume name for the unit remain PROGRAMS. You need only specify either of the following to get the same file.

```
#5:FILE
```

or

```
PROGRAMS:FILE
```

It is possible to change the working directory on an SRM or HFS unit without changing the default volume. Use the Filer's Unit directory command. Press ☐U and then give the directory name that you wish to become the working directory. If the new directory is in the existing working directory, just type the new directory name. If it is not, type the whole directory path as shown above in the Prefix example (or use the .. superior directory specifier and appropriate directory names).

Note

Do **not** use the Prefix command on unit #45 of #46. These are the system volumes for SRM and HFS, respectively, and should not be altered.

When Prefixing an HFS-formatted **flexible** disc, you are automatically prefixed to the root directory; you **cannot** prefix to any directory below the root level on this type of flexible disc. (However, with HFS-formatted **hard** discs, this restriction does **not** apply.)

The System Volume

The system volume can also be specified in a shorthand form using the * character. Suppose you want to specify the file named LIBRARY on the volume SYSVOL:. Assuming that SYSVOL: is the system volume and is currently in the disc drive associated with unit #3, you can specify a file on that volume by any one of the following three methods:

```
SYSVOL:LIBRARY
```
or
```
#3:LIBRARY
```
or
```
*LIBRARY
```

Of course you can make the specification even shorter by typing something like this:

```
*=ARY
```

However, if you are doing a critical operation, be sure that there are no other files on the same volume which fit that file specification or use the ? wildcard instead. If a file named GARY existed on SYSVOL:, the operation would also be performed on it. Once again, **use wildcards judiciously**.

Listing a Directory

To find out what files are on the disc called ACCESS:, press ☐L to initiate the List Directory command. The Filer prompts you to specify the volume whose directory is to be listed:

    List what directory ?

Respond by typing **ACCESS:** and pressing ⌈Return⌉ or ⌈Enter⌉. Notice that the colon (:) is part of the volume specification. The screen now displays the directory (catalog) for **ACCESS:**. It looks similar to this display:

```
ACCESS:             Directory type= LIF level 1
created   8-Oct-82  3.47.54 block size=256
 Storage order
...file name....     # blks     # bytes  last chng

FILER                  218       55808  8-Oct-82
EDITOR                 228       58368  8-Oct-82
LIBRARIAN              202       51712  8-Oct-82
MEDIAINIT.CODE         132       33792  8-Oct-82
TAPEBKUP.CODE           54       13824  8-Oct-82
FILES shown=5 allocated=5 unallocated=11
BLOCKS (256 bytes) used=834 unused=218 largest space=218
```

The name of the volume is displayed in the upper left-hand corner of the listing. To the right, the directory type is displayed. Pascal LIF discs have Level 1 directories. Level 1 directories contain the date the directory was created and the size of the volume. Level 0 directories do not. Your directory listing should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in storage order or alphabetical order. The size of blocks on LIF volumes is 256 bytes (1 sector). The size of blocks on WS1.0 volumes is 512 bytes and on HFS discs is usually 1024 bytes. The Shared Resource Management system does its accounting in 1-byte blocks. To have directories listed in alphabetical order, include [*] after the directory name. For example:

    LIFDIR:[*]

The column entries for each file include: file name, number of blocks used for storage, the file size in bytes, and the date the file was created or changed.

The last two lines display additional directory information including how many more entries can fit in the directory. The size of a LIF or WS1.0 directory is specified when the disc is initialized.

Getting a More Detailed Listing

To get a more detailed listing of the directory on a disc, press ⬚E⬚ (for the Extended Directory command) and you will be prompted for a volume name as before. Respond by typing:

ACCESS: ⬚Return⬚ or ⬚Enter⬚

Your screen now displays:

```
ACCESS:            Directory type= LIF level 1
created  8-Oct-82  3.47.54 block size=256
 Storage order
 ...file name....    # blks    # bytes    start blk ....last change...extension1
                type  t-code ..directory info... ....create date... extension2

FILER               218       55808          4 8-Oct-82 3.48. 6          0
               Code -5582                                                1
EDITOR              228       58368        222 8-Oct-82 3.48.18          0
               Code -5582                                                1
LIBRARIAN           202       51712        450 8-Oct-82 3.48.35          0
               Code -5582                                                1
MEDIAINIT.CODE      132       33792        652 8-Oct-82 3.48.44          0
               Code -5582                                                1
TAPEBKUP.CODE        54       13824        784 8-Oct-82 3.48.48          0
               Code -5582                                                1
< UNUSED >          218                    838
FILES shown=5 allocated=5 unallocated=11
BLOCKS (256 bytes) used=834 unused=218 largest space=218
```

The Extended directory listing contains all the same information as the List directory listing with additional information. It also contains the number of the block where the file starts (for LIF and WS1.0 discs), the file type as recognized by the file system, the type-code used by the directory system, SRM (or HFS) access information and two extension fields.

The "directory info" column shows the public access rights and the current file status for SRM/HFS files. Here are two examples of Extended directory listings; the first is of an HFS disc and the second of an SRM disc.

HFS Extended Listing

```
hfs11:       Directory type= HFS 755      17     9
changed 15-Mar-87  14.24.54 block size=1024
Storage order
 ...file name....    # blks    # bytes    start blk ....last change... extension1
                type  t-code ..directory info... ....create date... extension2

lost+found           8        8192          3 1-Jan-86 2. 4.14          2
               Dir     3 d755m    17u      9g                          -1
TOP                  1          96         64 8-May-86 8. 9.53          3
               Dir     3 d755m    17u      9g                          -1
just_a_file          8        7168          4 15-Aug-86 6.10.24         1
               Data -5622 644m    17u      9g                        7618
FILES shown=3 allocated=3 unallocated=18497
BLOCKS (1024 bytes) used=17 unused=21726
```

The access information for HFS files and directories is given in the form of an octal number (a number of base 8). In the listing above, the two directories, lost+found and TOP, have access rights defined in the "directory info" column by d755m. The "d" shows that this file is in fact a directory and the octal number is 755. For the file called just_a_file, the access rights are given by the number 644. For a detailed explanation of these codes and the other information in the "directory info" column see the Extended directory and Hfs command later in this chapter.

SRM Extended Listing

```
MYDIR:             Directory type= SRM  21,0,8
created 20-Jun-86 10.45.52 block size=1
changed 11-Sep-86 17. 6.10 Storage order
...file name....  # blks    # bytes   start blk ....last change... extension1
                  type  t-code ..directory info... ....create date... extension2

SOMEDATA.TEXT        12288       12288             20-Jun-86 10.45.56        -1
                Text   -5570 MRWSPC CLOSED         20-Jun-86 10.45.56        -1
Bigfile               2048        2048             12-Aug-86 12.54. 9        -1
                Data   -5622 MRWSPC CLOSED         12-Aug-86 12.54. 5        -1
FILES shown=2 allocated=2
BLOCKS (1 bytes) used=14336 unused=14496768
```

If one of the letters from the table below is missing, then the public access right associated with that letter has been removed. (Not applicable to HFS discs).

Letter	Access Right
M	Manager
R	Read
W	Write
S	Search
P	Purgelink
C	Createlink

Public access rights on a file are established at one of two times. If a file is created by a program, the public access rights can be established when the file is opened. To do this, use the optional third parameter on the command used to open the file. The commands used to open files are: Reset, Rewrite, Open, and Append. The optional third parameter is explained in more detail in the Programming with Files chapter in this manual.

If a file already exists, the Filer's Access command can be used to establish or, if the Manager right has not been removed, change the public access rights.

The possible "current file status" are listed below and explained in the Programming with Files chapter.

CLOSED   SHARED   EXCLUSIVE   CORRUPT

The two extension fields are for LIF and HFS directories. For most LIF file types on a LIF disc, extension1 contains a "0". For system files, it contains the start execution address. For data files, it contains the logical end-of-file. Extension2 contains the volume number in cases of multi-volume files. The Pascal system cannot create or read multi-volume files; the LIF DAM merely recognizes them. For single volume files, it contains a "1". HFS assigns different meanings to the extension fields; see "HFS Listing Information" near the end of Chapter 3.

The above examples are the most common uses of the directory listing commands, but there are two other useful ways of using the command. One is to use a "wildcard" to specify a subset of files that you want listed. The other way is to send the listing to the printer or to a file instead of letting the listing default to the screen. Both methods are combined in the example below and are covered in detail in the "Filer Commands" section. Press ⎡E⎤ again (to initiate the Extended Directory command) and answer the prompt for a volume specification as shown in the display:

```
List_ext what directory ? ACCESS:=.CODE,PRINTER:  [Return] or [Enter]
```

The ACCESS: volume should be on-line. Your specification tells the Filer you want a listing of all the files on the ACCESS: disc whose name ends in ".CODE". The "=" acts as a substitute for all combinations of characters in a file name. The "," separates the source file specification from the destination file specification. The listing will only display the files whose names end with ".CODE". The EDITOR, FILER and LIBRARIAN are not listed because their names don't end in ".CODE".

A Few Words About Wildcards

Wildcards are powerful tools for executing Filer commands on related files. There are three wildcard characters.

```
?        =        $
```

A wildcard is a substitute for an arbitrary portion of a file name. For example, if you wanted to list all the .CODE files on the EXAMP: volume, you could specify:

```
EXAMP:=.CODE
```

The "=" stands for any combination of characters. If the file name ended with ".CODE", that file would appear in the listing. If you wanted to remove some of the .TEXT files on the EXAMP: volume, you could specify:

```
EXAMP:?.TEXT
```

The "?" also stands for any combination of characters. However, the Filer will ask you, one at a time, if you want to remove each file if it fits the specification. The "?" wildcard lets you verify operations before actually performing them. Unless you are absolutely certain about the effects of a command using the equals sign wildcard (=), it is best to use the question mark — by far the safer of the two.

The "$" character is a valid wildcard for destination file specifications. It indicates that the file is to retain its original name. If "$" is used with other characters, the "$" is used as part of the name.

Wildcards act as replacement strings in file names. Part of a file name can be given before or after the wildcard or both before and after. For example, two files named WILD.TEXT and WILD.CODE on the default volume could be specified by:

WILD? or WILD= or =LD.= or ?ILD=

If you specify more than one partial file name, they must be given in the order in which they appear in the file name.

Translating Text Files

The Pascal system supports several different types of "text" files. These files are usually created by the Editor and can be programs, documents, or data. When the file is stored on a disc, the internal representation of data in the file (i.e., file type) is determined by the suffix appended to the file specifier when the file was created. The different file types have different information in the file header and can have different end-of-line schemes. The Translate command can be used to convert files (that contain textual data) from one file type to another. The file types that can be used to store textual data and that can be recognized by the Pascal system are TEXT, ASCII, HP-UX "compatible", and Data. A .TEXT suffix indicates a TEXT file, a .ASC suffix indicates an ASCII file, a .UX suffix indicates a file used for data exchange with a system running under HP-UX, and the absence of a recognized suffix indicates a Data file.

To use the Translate command, press T and see the prompt:

```
Translate what file ?
```

Respond with the name of your input file

MYVOL:MYFILE.TEXT

The Filer will then prompt

```
Translate to what ?
```

Respond with the name of your output file

MYVOL:NEWFILE.ASC

The Filer will create an output file of a type corresponding to the suffix on the output file name (.ASC in the example) and will read the text data from the input file, reformat the data to match the output file type, and write the data to the output file. This process may seem slow, but remember that the text is being reformatted.

Sending File Listings to the Printer and Screen

The Translate command is also used to send files to the printer or to the screen. Logically, the printer and screen are just files of a different format.

Before using the Translate command, remove the ACCESS: volume and replace it with the DOC: volume (supplied with this manual set). Now use the Extended Directory command to display the contents of the DOC: volume. Press ⌧E⌧, type in DOC: and press ⌧Return⌧ or ⌧Enter⌧. Your screen should display all the files on the documentation disc.

Press the spacebar: this clears the screen of everything except the Filer prompt. Now press ⌧T⌧ to initiate the Translate command. The screen prompts you with:

```
Translate what file ?
```

Respond with DOC:BINDOC.TEXT and press ⌧Return⌧ or ⌧Enter⌧. The screen now prompts:

```
Translate what file ? DOC:BINDOC.TEXT
Translate to what ?
```

Your first response included both a volume specification and a file name and completely identifies the file you want to transfer. This time, however, type only the PRINTER: volume specification and press ⌧Return⌧ or ⌧Enter⌧. The text file is translated to the printer as shown:

```
BEGIN {Binery_search}
  done:=FALSE;  btm:=0;  top:=26;   {initialize}
  FOR loop:=1 TO top DO alpha[loop]:=CHR(loop+64);
  WRITELN('Type uppercase character for a key');
  READ(key);  WRITELN;
  WHILE NOT done DO
  BEGIN          {This is the actual binery search}
  mid:= ROUND((top + btm)/2);
    IF key = alpha[mid] THEN done:= TRUE
    ELSE IF key < alpha[mid] THEN top:=mid
    ELSE btm:=mid;
    IF top=btm THEN BEGIN
                    done:=TRUE;  mid:= -1;
                  END;
  END;
  IF mid > 0 THEN
   WRITELN('Key -',key,'- is in position ',mid:2)
  ELSE WRITELN('key - ',key,' - was not found');
END.
```

The Filer shows you what operation it has just performed by displaying:

```
DOC:BINDOC.TEXT                    ==> PRINTER:
```

Since the operation is complete, the Filer again displays its prompt. Note that only files of type TEXT, ASCII, UX, or Data should be sent to the printer. You can also Translate these files to the screen by using CONSOLE: in the destination specification instead of PRINTER:. The file is displayed one screen at a time. Press the spacebar to move to the next screen; press [Shift]-[Select] ([Shift]-**EXECUTE**) to abort the operation.

The Translate command can also be used to direct files to the SRM printer on an SRM system. Using a shared printer or plotter to output data requires you to place your data in an ASCII file in the spooler directory. Once a file is in the directory, the SRM operating system sends the file to the appropriate output device as soon as the device is free. See the Translate command for more details.

If you are not sure if the file in question is a text file, use the Extended Directory command and look at the column in the display where the file types are shown.

Copying Entire Volumes: Backup Copies

The backup process described here is suitable for volume-to-volume copies if both volumes are the same size. For different size volumes, see Filecopy in the "Filer Commands" section.

Note

Using Filecopy to copy an entire volume will result in the loss of disc space if the source volume is *smaller* than the destination volume. To copy a volume to a larger one, Filecopy individual files.

You should still be at the Main Command Level and now have a blank initialized disc. We will use it for a volume-to-volume Filecopy. Volume-to-volume Filecopy operations do not require that a directory be present on the destination disc, but does require it to be initialized.

Insert ACCESS: in the disc drive. Press [F] and the Filer will be loaded and display its prompt:

```
Filer: Change Get Ldir New Quit Remove Save Translate Vols What Access Udir ?
```

Press [F] for the Filecopy command and the screen shows:

```
Filecopy what file ?
```

Now type **ACCESS:** and press [Return] or [Enter]. The screen displays:

```
Filecopy what file ? ACCESS:
Filecopy to what ?
```

You can specify a volume by specifying the logical unit number associated with the physical disc drive that it is in. Do this by typing **#3:** and pressing [Return] or [Enter]. The Filer knows that ACCESS: is currently in the drive associated with unit #3 and figures that you want to transfer that volume to a different volume that will be inserted in the same drive. The Filer then reads as much of ACCESS: as it can into read/write memory and the screen displays:

```
Please mount DESTINATION in unit #3
'C' continues, <esc> aborts
```

Now remove ACCESS:, replace it with the blank initialized disc, and press [C]. Since no directory is on the initialized volume, the Filer simply copies the ACCESS: information that it read into memory onto the new disc. If there had been a directory named TESTER: on the destination volume, the Filer would have prompted:

Destroy EVERYTHING on volume TESTER: ? (Y/N)

This precaution makes sure the information on the disc does not get destroyed if you change your mind or inserted the wrong disc. Answering with [N] for "No" aborts the Filecopy operation and the Filer prompt returns. Answering with a [Y] for Yes lets the Filecopy take place; the contents of ACCESS: are written to the new disc. This operation destroys the directory (and, effectively, all information) that was previously on the destination disc.

In case your machine does not contain enough memory to read in the entire volume ACCESS:, the Filer prompts you to swap the source and destination discs as many times as necessary to complete the Filecopy operation. When the operation is complete the Filer prompt reappears.

If you have more than one disc drive you can accomplish the same task by specifying both the source and destination volumes with either a volume name (if it has one) or by the unit number associated with the drive it is in. This second method of doing volume-to-volume transfers is quicker — especially if the amount of memory in your machine is relatively small.

Note

Having two volumes with the same name on-line at one time is not advised. The Filer looks for volumes according to their volume names and may not be able to distinguish one from the other. Thus, the Filer may perform an action on one volume when you wanted the operation to affect the other volume. The Filer warns you whenever it detects that this condition exists. If you get a warning, either remove one of the volumes or use the Filer's Change command to change the name of one of the volumes. (Specify it by unit number rather than by volume name, or take the other volume of the same name off-line during the Change operation.)

Creating a Directory

The Filer generally only works with volumes that have directories. There are a few exceptions to this such as volume-to-volume transfers, where the directory from the source volume is copied onto the destination volume. Other exceptions are mentioned as they arise. The Filer's Zero command creates an empty directory on a new disc that has been initialized using the MEDIAINIT program, previously used discs, or on any other compatible type of mass storage device such as a hard disc or a volume stored in read/write memory. The Zero command, however, is not used to create directories on the Shared Resource Manager or on HFS discs. This is done with the Make command because making an SRM/HFS directory really involves making a file of type "Directory". Also, for HFS, there is a special utility to create the root volume of an HFS disc. This is called MKHFS and its operation is covered in Chapter 21, "HFS Set-Up and Utilities", in Volume II of this manual.

Your screen should now display the Filer prompt. Remove the current volume from the disc drive associated with unit number 3 and replace it with the second disc that you initialized. Now press ⌷Z⌷ to initiate the Zero command. The screen displays:

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

The request is for a volume specification. Answer with #3: and press ⌷Return⌷ or ⌷Enter⌷. The Filer now prompts:

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

This question is just a safety precaution so that you won't destroy a volume full of information by accident. "V3" is the name given to the directory by MEDIAINIT (if created on unit #3). Press ⌷Y⌷ for yes. The next prompt is:

```
Number of directory entries (80) ?
```

This is asking for the maximum number of files that will be listed in the directory. The number in the parentheses is the default that will be used if no value is given and is derived from the number in the existing directory. In most cases, 80 directory entries is a good choice.

The next prompt is:

```
Number of bytes (270336) ?
```

This is asking for the total size of the disc to be handled by the directory (the logical size of the volume). The number in the parentheses is derived from the number in the existing directory (if any) or from the unit table entry for that given unit. Press ⌷Return⌷ or ⌷Enter⌷ to accept the default size for your disc.

The system now prompts you for a volume name. Volumes and volume name syntax for the different directory types are described in the File System chapter. Briefly, LIF directory names must be six characters or less, uppercase and lowercase letters being distinct. WS1.0 directory names must be seven characters or less, and are always uppercased before being written in the directory. The Filer then confirms that the volume name is the one you wanted.

The screen now appears:

```
Zero directory (NOT valid for HFS and SRM type units)
Zero what volume ? #3
Destroy ACCESS: ? (Y/N) Y
Number of directory entries (80) ? 80
Number of bytes (270336) ?
New directory name? NEWONE
NEWONE: correct ? (Y/N)
```

When you press Y to confirm the new volume name the Filer informs you that the volume with that name has been zeroed and the Filer's prompt appears. Your new volume is now ready for use.

Copying Files from Volume to Volume

The Filecopy command allows you to copy files from one volume to another or even to a different place on the same volume. The volumes can be separate discs, SRM directories or, in the case of a hard disc, multiple volumes on the same physical device.

Remove the current volume from drive #3 and insert the DOC: volume supplied with this document set. To copy a file from one volume to another, press F for Filecopy and respond to the prompt for a file specification with:

```
DOC:STREAM.TEXT
```

When the Filer prompts you for a destination, type in the specification shown below and press Return or Enter.

```
Filecopy what file ? DOC:STREAM.TEXT
Filecopy to what ? #3:$
```

What happens here is similar to copying a volume from one disc to another using a single drive. The Filer reads the contents of DOC:STREAM.TEXT into memory and then displays the message:

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

Take another disc and insert it in drive #3. Now that you have your new disc in drive #3, press ⎡C⎤ to continue. The Filer writes the contents of the file that it temporarily stored in memory to the disc you just inserted and confirms that the Filecopy has taken place.

If you give a unit number (as above) or a different volume name which is not on-line, you must swap discs to complete the copy.

The wildcard ($) is a feature to avoid repetitious typing; it tells the Filer to give the destination file the same name as the original file — **STREAM.TEXT**.

When copying a file to a different volume, **always** include either a file name or the **$** character when you specify the destination. If you specify the name of a mass storage volume without a file name, the Filer prompts with a message like this:

```
Destroy EVERYTHING on volume ⟨volume name⟩ ? (Y/N)
```

Although the volume name may be different, if you answer with a ⎡Y⎤, the Filer transfers the specified file to the destination volume, **destroying** the directory in the process, and rendering all previous information on that volume useless.

The next example demonstrates how to copy multiple files from one volume to another using the ? character as a wildcard. Press ⎡F⎤ once again and respond to both the prompts as shown:

```
Filecopy what file ? DOC:MOD?TEXT
Filecopy to what ? MKWORK:$
```

This tells the Filer to copy all the files on the DOC: volume that begin with the characters "MOD" and end with the characters "TEXT" to the volume MKWORK:, giving them the same name it had on the DOC: disc. Before the Filer actually copies any files, however, it will verify with you that you actually want to copy each file that fits the specification. Respond to each prompt with a ⎡Y⎤ for "Yes". As you answer each prompt affirmatively, the Filer copies the corresponding file to the destination (MKWORK:) volume. If you have a single-drive system, the Filer will prompt you to swap the discs as in the previous example.

It is worth mentioning that, although specifying a unit number is less typing than specifying a volume name, when you specify a unit number the Filer initially accesses the volume (disc) currently in the drive without regard to whether or not it was the one you intended. After the first access of a volume, the Filer associates a supplied unit number with the name of the volume found in that device. However, if you specify a volume name, the Filer only performs the command on that volume. If the volume you specified is not on-line the Filer will tell you so. Specifying the volume name is a good habit if you are doing a lot of disc "swapping"; this will insure that the Filer does not operate on a disc other than the one you intended to use.

In cases where the destination volume already contains a file with the same name as the file being copied, this prompt is displayed:

```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

You have the options:

- Remove — remove the original file first, then write the new file in the largest space available.

- Overwrite — replace the contents of the old file with the new information. The Overwrite option cannot be used to change the *type* of a file. Attempting to do so would result in the file contents being inconsistent with the file type.

- Neither — cancel the operation.

The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM and HFS disc users when duplicate links exist to a file. All links and access rights to the file are accurate when a file is updated because it is put in the same logical location. If you chose the Remove option, and the file has other links, the original file would not actually be removed; only your link to it is removed. The other users' directories are still linked to the original file.

Note

Be careful when using the Overwrite option. If the file type is not the same as the original file type, the contents of the file may become inaccessible.

Care should be exercised when using the Overwrite option. Consider the following example:

You have a file called **ABC** which you intend to use to overwrite another file called **XYZ.TEXT**. File ABC is of type Data and file XYZ.TEXT of type Text. If this particular example is carried out, the following will occur:

- The file type of XYZ.TEXT will be retained, even though the file contents are overwritten.

- The contents of ABC will be copied into the location where the contents of XYZ.TEXT reside, overwriting as the operation is executed.

This will result in a file called **XYZ.TEXT** which has a file type Text, but whose contents are in fact of type Data. If the Editor, or another subsystem, is used to try to access this file an error will almost certainly occur. This is because the system assumes that the file *contents* will conform to the format expected of a file of type Text, and in reality this is false.

Note

Never use Overwrite when replacing SYSTEM files (**.SYSTM** suffix). Some necessary information is not updated by Overwrite.

Renaming Files and Volumes

The Filer's Change command allows you to rename files and volumes. (The exceptions are that the root directory of the Shared Resource Manager cannot be renamed, neither can the name of an SRM or HFS directory which is the Unit directory. These may be the same in certain circumstances.) This command requires two specifications: the original name and the new name. (The original name may include volume specification, directory path for HFS and SRM, and passwords for SRM; however, the new name **cannot**). Assuming that the volume MKWORK: is still on-line, press ⌜C⌝ for Change and respond to the prompt as shown:

```
Change what file ? MKWORK:,MOJO:
```

The volume name is now "MOJO:". To change the file STREAM.TEXT on the MOJO: volume to RIVER.TEXT you can either type out both names (separated by either a comma or a press of the ⌜Return⌝ or ⌜Enter⌝ key) or use a wildcard as shown below:

```
Change what file ? STREAM=
Change to what ? RIVER=
```

The Filer changes the file name as described. The wildcard was used as a substitute for the .TEXT part of both names. The only restriction on using wildcards with this command is that if you use a wildcard in one of the specifications, you must use it in the other. Because the strings or subsets represented by the wildcard are not always obvious, discretion is advised when using wildcards with the Change command.

When changing the name of a file of type TEXT or CODE, remember that parts of the Pascal System attempt to append the suffixes ".TEXT" or ".CODE" to the file you specify. You can get around this by specifying a file and adding a period (.) to the file name. This tells the system not to append a suffix to the file name.

Note

Excluding the Get command, the Filer makes no assumptions about suffixes and will treat a trailing period as part of the file name.

Removing Files

The Remove command is provided to delete files from a directory of a block structured volume. Suppose you have a volume on-line named NEWSTUF: containing the file POLYNOM.TEXT that you wish to delete. Press ⌈R⌉ to initiate the Remove command and respond to the prompt as shown:

```
Remove what file ? NEWSTUF:POLYNOM.TEXT
```

Then press ⌈Return⌉ or ⌈Enter⌉. The Filer removes the specified file from the volume and reports:

```
NEWSTUF:POLYNOM.TEXT removed
```

The Filer prompt reappears as the message is displayed. As in many of the Filer's commands, the prompt requests a file specification. Wildcards can be used with the Remove command but should be used carefully. The question mark (?) wildcard provides an easy method for removing a TEXT and CODE file of the same name. It also lets you verify the operation (a good practice when purging files).

Suppose the same volume NEWSTUF: contains two files you wish to remove, and they are called: IOTEST.TEXT and IOTEST.CODE. To remove these files answer the "Remove what file ?" prompt with:

```
NEWSTUF:IOTEST?
```

and press ⌈Return⌉ or ⌈Enter⌉. The Filer responds with:

```
Remove IOTEST.TEXT ?   (Y/N)
```

Reply with ⌈Y⌉ (for Yes) to remove the file. Reply with ⌈N⌉ (for No) if you change your mind. Either reply results in the next prompt:

```
Remove IOTEST.TEXT ?   (Y/N) Y
Remove IOTEST.CODE ?   (Y/N)
```

Reply as before and the Filer responds with:

```
Remove IOTEST.TEXT ?   (Y/N) Y
Remove IOTEST.CODE ?   (Y/N) Y
Proceed with remove ?  (Y/N)
```

This gives you one more chance to change your mind about the operation. The files are not actually removed from the volume's directory until you press ⌈Y⌉. Pressing ⌈N⌉ has the same effect as if you had never initiated the command (i.e., the directory remains unchanged and your files remain intact).

If you want to remove all of the files on a volume (for non-HFS local discs only, not SRM), the quickest way to do so is to execute the Zero command. This command wipes out the directory of a volume so that the volume may be re-used. See the description of the Zero command earlier in this chapter or in the "Filer Commands" section.

Leaving the Filer

Exit the Filer by pressing $\boxed{\text{Q}}$ for Quit from the Filer prompt. You will immediately be returned to the Main Command Level. The Filer can also be exited with the $\boxed{\text{Stop}}$ key. The $\boxed{\text{Stop}}$ key waits for any current disc I/O to complete before it actually executes. This key can be used at any time — even while executing a Filer command. However, this practice is not recommended since some commands may cause damage to your files if $\boxed{\text{Stop}}$ is pressed while they are being accessed.

The System Workfile (A Convenient Scratchpad)

The Pascal System features a workfile which can be used in the Filer, Editor, Compiler, and Assembler. Using workfiles with each subsystem is documented in the corresponding chapter of this manual.

Think of the workfile as being analogous to a default volume. In some of the subsystems, you are not prompted for a file specification when entering the subsystem if a workfile of the appropriate type exists. For example, if the text version of a workfile exists when entering the Editor, the Editor never prompts you for a name of the text file to edit but reads in the workfile instead. As a matter of fact, before you can edit any other file, you will need to use the Filer's New command to release the workfile (preceded by the Save command if you want to store the file). In the same manner, invoking the Pascal Compiler when the text version of a workfile exists (but not the code file) results in that file automatically being compiled.

If the Filer's Get command is used, the workfile is both the source (TEXT, ASCII, UX, Data) and object (CODE) file specified in the command.

The Filer has four commands (Get, New, Save, and What) which operate directly on the workfile. These are covered in the next section.

Filer Commands

This section contains a brief overview and summary of the Filer commands and a complete alphabetized description of the syntax and semantics of all the Pascal Filer commands and options.

Filer Command Summary

Volume Related Commands

Bad sectors   Scans a local medium 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, SRM/UX, 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, SRM, and SRM/UX only)

Zero   Creates an empty directory on the specified volume. (Not valid for HFS, SRM/UX 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, SRM, and SRM/UX 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 or SRM/UX volume.
Make	Create a directory (HFS, SRM, and SRM/UX) 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).

Command Syntax and Semantics

The Filer commands are presented in alphabetical order. Each command's explanation includes: the command's name, a brief functional description, a diagram showing its legal syntax (see Chapter 2 for an explanation of the syntax used), the command's prompt (if any) and text which discusses using the command. Each command's options are also covered and some have examples to show the proper use of these options.

Several of the syntax diagrams on the following pages reference the "volume specification" and the "file specification" on the following pages. The "volume specification" is the syntax for commands that operate on volumes. The "file specification" is the syntax for commands that operate on files. Volume specifications don't need the ":" except when a literal volume name is given. Then the name must end with a ":" to distinguish it from a file name. If no volume specification is given, the default volume is assumed.

Alphabetical List of Filer Commands

Access
Bad sectors
Change
Duplicate
Extended directory
Filecopy
Get
Hfs
Krunch
List directory
Make
New
Prefix
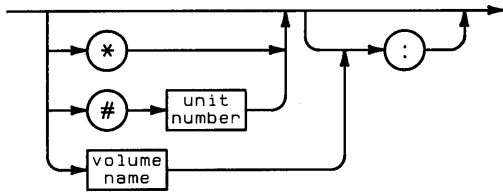Quit
Remove
Save
Translate
Unit directory
Volumes
What
Zero

File Specification
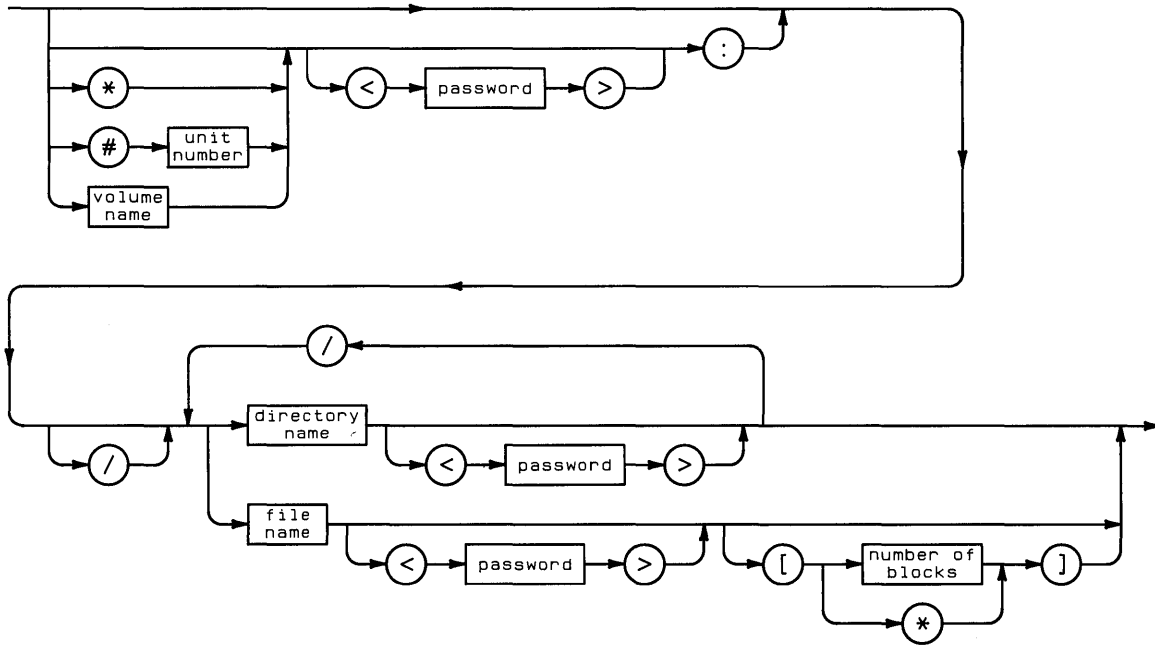
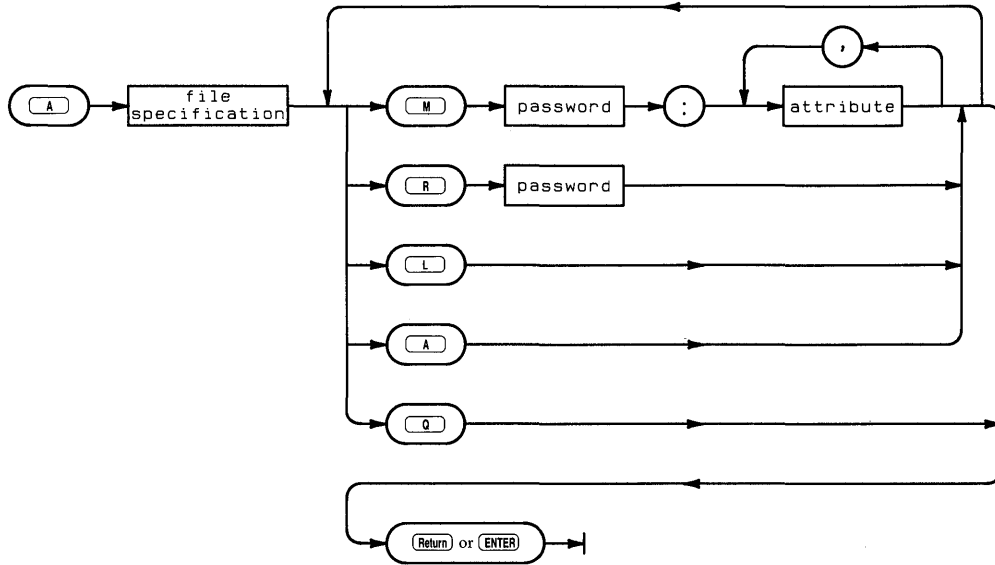| Item	Description	Range
unit number	integer; corresponding to an entry in the Unit Table	1 through 50
volume name	literal	any legal volume name
password	literal	any legal password (SRM only)
directory name	literal	any legal SRM, SRM/UX or HFS directory name
file name	literal	any legal file name
number of blocks	integer	any legal number of blocks

See Chapter 3 for legal names and values.

Volume Specification

Item	Description	Range
unit number	integer; corresponding to an entry in the Unit Table	1 through 50
volume name	literal	any legal volume name

Access

The Access command allows you to change public access rights on your files (SRM only).

Item	Description	Range
file specification	literal	a legal SRM file specification
attribute	literal	MANAGER READ WRITE SEARCH PURGELINK CREATELINK ALL
password	literal	any legal password (See Chapter 3 for details)

Semantics

All access capabilities for a file are initially public. You can remove one or more capabilities from public access by associating them with password(s).

The Access prompt:

```
Access codes for what file ?
```

Type the file specification. If the file already has a MANAGER password, then you must include the password in the file specification.

The next prompt:

```
Access: List, Make, Remove, Attributes, Quit ?
```

These are the possibilities. You can list the attribute passwords, make new ones or remove passwords. The Attributes option just lists the possible attributes for your help. Quit returns you to the Filer prompt.

Access rights cannot be changed on open files or open working directories.

To make new passwords, press ⌷M⌷ . You see this prompt:

```
Make password:attribute ?
```

Type the password (up to 16 characters), then a colon (:) followed by the attribute list (with attributes separated by commas). Different passwords may be associated with each attribute or one with ALL. If you type a password that already exists, you are asked:

```
PASSWORD already exists...replace it ? (Y/N)
```

Note that passwords should not contain these three characters: > : , .

To remove passwords, press ⌷R⌷ . You see the prompt:

```
Remove password ?
```

When you type only the password, all attributes associated with it are cleared.

The Attributes option list:

```
MANAGER
READ
WRITE
SEARCH
PURGELINK
CREATELINK
ALL
```

Manager - permits the user to assign or remove further attributes.

Read - permits the user to read the contents of the file.

Write - permits the user to write, or overwrite data to or in the file.

Search - permits the user to look at the contents of a directory. This is really a special form of *Read* which is applicable to directories.

Purgelink - permits the user to purge links to the file.

Createlink - permits the user to set up new links to the file.

All - associates the given password with all the above listed attributes.

Depending on the attributes associated with a particular file, it may not be permitted to perform some file or directory operations using certain Main- level or Subsystem commands. As an example, let us suppose you had a file which you wanted a group of people to have access to (but not all those using the SRM system), but you wished to be the only person who is permitted to write or overwrite the file. First you would give a password and the READ attribute. This is the password you will tell the group of people. Second, you would give a different password and the WRITE and MANAGER attributes. This password only you would know. This would give the desired protection to the file.

Bad sector

The Bad Sector command scans a local mass storage medium for errors. It is not valid for SRM or SRM/UX.

```
( B )──▶ volume        ──▶( Return ) or ( ENTER )──▶|
        specification
```

Item	Description	Range
volume specification	literal	(See the beginning of this section)

Semantics

The Bad sector prompt:

    Bad sector scan of what directory ?

The Bad sector command allows you to check a mass storage medium to find out if each block (sector) is readable. Mass storage media may become unreliable after damage or excessive wear.

Press ⬚B⬚ to initiate the command and answer the prompt with a volume specification. The Filer then displays a message indicating that it is scanning the volume from block 0 to the end of the volume. The Filer does a read operation on each sector and if the read succeeds, that sector is considered to be good; if not, the Filer lists the sector number.

If you find a bad sector in a file, you may wish to use the Filer to change the file type (suffix) to .BAD. (You did make a back-up copy didn't you?) The BAD file will not be moved in a Krunch operation. A large number of bad sectors indicates worn-out media. The media should only be used if you are willing to risk losing information on that volume.

Change

The Change command lets you rename files, directories, and volumes.
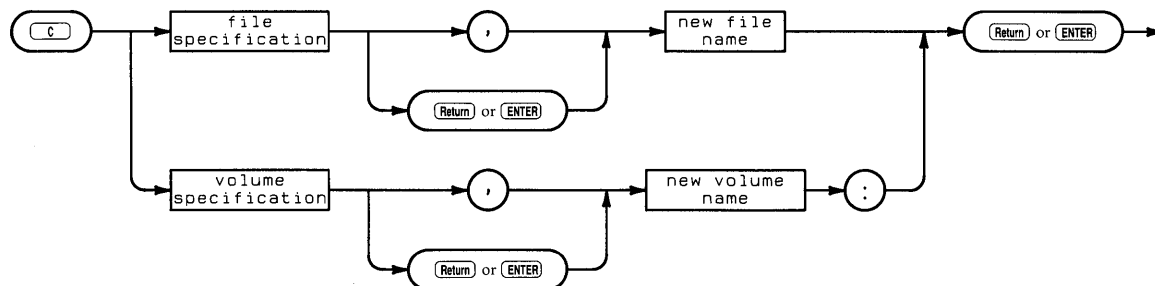
Item	Description	Range
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)
new file name	literal	any valid file name
new volume name	literal	(See the beginning of this section)

Semantics

The Change prompt:

  Change what file?

The Change command requires two specifications: the original volume or file specification and the new one. The two specifications can be separated by either a comma or a carriage return.

To change the name of a file, use any legal volume ID in the first specification and only the new file name in the second specification. The Filer is intelligent enough to know that the file whose name you are changing resides on the volume identified in the first specification. After the Filer has finished changing the name(s) and updating the directory, it reports the name change(s) on the CONSOLE: volume.

Because many of the Pascal subsystems append the string .TEXT or .CODE to a file name given in response to a prompt, it is a good idea to retain these parts of a file name when making a change.

Wildcards (the = and ? characters) may be used in the Change command. If a wildcard is used in the first specification, it must also be used in the second one. The subset string that is replaced by the wildcard in the second specification (the new name) is the same as the string it stands for in the first specification.

Suppose you have a volume named BUGS: with the following files:

```
WHATISIT.TEXT
WHOISIT.TEXT
WHYISIT.TEXT
```

Specifying BUGS:WH=TEXT, FO=FA in response to the Change prompt results in the following messages being reported by the Filer:

```
BUGS:WHATISIT.TEXT  changed to FOATISIT.FA
BUGS:WHOISIT.TEXT  changed to FOOISIT.FA
BUGS:WHYISIT.TEXT  changed to FOYISIT.FA
```

Here is another example using the files shown above on the BUGS: volume. Specifying BUGS:WH=.TEXT, = results in:

```
BUGS:WHATISIT.TEXT changed to  ATISIT
BUGS:WHOISIT.TEXT  changed to OISIT
BUGS:WHYISIT.TEXT  changed to YISIT
```

You may wish to create some empty files using the Make command and experiment with them before using wildcards extensively. Until you get used to them, the effects of wildcards are not always obvious.
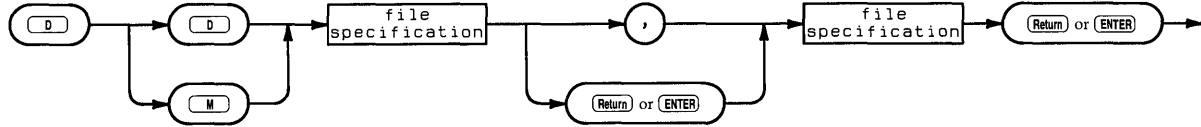

Caution

Possible loss of data. Using the Change command to "change" a file name to the same name results in the file being removed.

Note

The Change command does **not** change the file type. (See Translate.)

Duplicate

The Duplicate link command establishes a new pointer to a file (HFS, SRM, or SRM/UX only).

Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Duplicate link prompt:

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

Do you want the original pointer to the file removed after the duplicate link is established? If you do, type ☐M☐ for Move; if not, type ☐D☐ for Duplicate.

If you choose the Duplicate option, the next prompt is as follows:

```
Dup_link what file ?
```

Type the file specification (including the SRM password if the CREATELINK capability has one).

```
Dup_link to what ?
```

Type the new file specification. Wildcards can be used in the specification. This puts a link to the file in a second directory.

If the file is referenced from two or more directories, the file is physically removed from the disc only when all links to the file have been removed.

You should be aware that new CODE files generated by the Compiler, Assembler, and Librarian to replace older versions are not written in the same space. If several directories have duplicate links to the same CODE file and the CODE file is recompiled, only one directory has an accurate link to the **new** CODE file. Other users must use the Duplicate link command to become linked to the new CODE file.

Caution

Possible loss of data. Using the Duplicate command to "duplicate link" or move a file to the same file results in the file being removed.
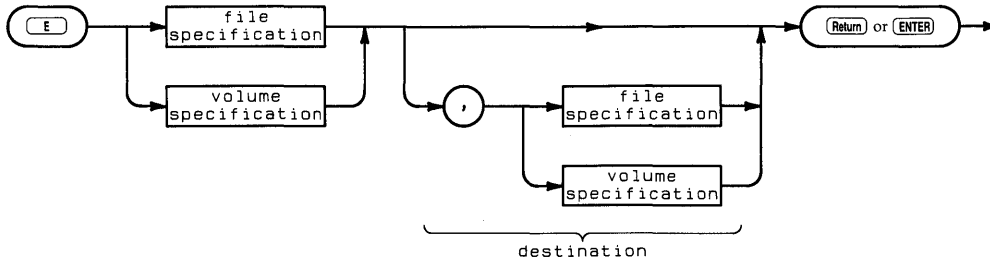

If you choose the Move option, the next prompt is as follows:

`Move what file?`

Type the new file specification. Wildcards can be used in the specification. This puts a link to the file in a second directory, and the original link is then removed.

Extended directory

The Extended directory command lists the directory of a blocked volume or a set of files in the volume.

Item	Description	Range
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Extended directory prompt:

```
List_ext what directory ?
```

The Extended directory command requires a legal volume or file specification. Results can be listed to the PRINTER: or to a text file if specified and separated from the first specification by a comma. If no destination is specified the listing defaults to the CONSOLE. Wildcards are available to identify subsets of files on the volume.

In the listing, the name of the volume is displayed in the upper left-hand corner. To the right, the directory type is displayed. Pascal LIF discs have Level 1 directories; level 1 directories contain the creation date and volume size information. Level 0 directories (created on other systems) do not. Your directory listing should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in **Storage order** or **Alphabetic order**. The size of blocks on LIF volumes is 256 bytes. The size of blocks on WS1.0 volumes is 512 bytes and on HFS volumes is usually 1024 bytes. The Shared Resource Management system uses single byte "blocks".

To have directories listed in alphabetical order, include [*] after the directory name. For example:

```
MYDIR: [*]
```

The column entries for each file depend on the type of directory being cataloged (LIF, SRM, SRM/UX, or HFS) but can include: file name, number of blocks used for storage, the file size in bytes, the number of the block where the file starts, the date the file was changed, the type as recognized by the file system, the type-code used by the directory system, SRM/HFS access information, the date the file was created, and two extension fields.

The SRM access information column comes under the heading "directory info". It contains codes which show the access rights which are still public (i.e., which have not been protected by associating them with a password):

M   Manager

R   Read

W   Write

S   Search

P   Purgelink

C   Createlink

And the current file status:

CLOSED
SHARED
EXCLUSIVE
CORRUPT

CLOSED, SHARED, and EXCLUSIVE are file status that are associated with SRM systems and are explained in detail in Chapter 15. If a file is ever marked CORRUPT, your Shared Resource Manager has a problem. Stop your operation and notify the person responsible for your SRM. He or she should restore the SRM to a usable state.

The last two lines display additional directory information including how many more entries can fit in the directory or on the file system as a whole. (The available number of entries in a directory was specified for LIF when the disc was initialized.)

The results can be listed to a printer or a file if you so specify. The destination of the listing is separated from the volume specification or file specification being listed by a comma. If no destination is specified, then the listing defaults to the screen. Wildcards are available to specify groups or subsets of files on a mass storage medium.

For example, assuming that SYSVOL: (the system volume) is in the disc drive which has been assigned logical unit #3, a listing of all the CODE files on that volume could be sent to the printer by specifying any of the following in response to the Extended directory prompt:

`#3:=CODE,#6:`   specifies volume residing in unit #3; listing to logical unit #6: (the PRINTER: volume)

`*=CODE,PRINTER:`   specifies system volume; listing to the PRINTER. (Without the colon, the listing would be sent to a DATA file named "PRINTER" on the default volume.)

`SYSVOL:=CODE,#6`   specifies SYSVOL: volume; listing to unit #6.

In all cases the "=CODE" string refers to all files whose names end in CODE on the specified volume and the listing is sent to the printer.

Listings can also be sent to a file. Use a destination parameter after the source parameter (separated by a ",") as in the above PRINTER: example. Give a complete file specification, using the appropriate suffix in the file name, otherwise a file of type Data is produced. For example:

```
List what directory ? #3:,SYSVOL:LIST.TEXT
```
or
```
List what directory ? #3:,SYSVOL:LIST.ASC
```

If the directory being listed is of type HFS, the "directory info" column is also used to display the currently assigned HFS access rights. The column is subdivided into three further columns, giving information about the mode, user and group respectively. This information is supplied in the form of octal numbers. Let's take a look at an extended listing of an HFS directory.

```
hfs11:        Directory type= HFS 755      17       9
changed 15-Mar-87  14.24.54 block size=1024
 Storage order
...file name....    # blks    # bytes    start blk ....last change... extension1
                  type  t-code ..directory info... ....create date... extension2

lost+found               8        8192         3 15-Mar-87  2. 4.14         2
             Dir       3 d755m      17u    9g                            -1
TOP                      1          96        64  8-May-86  8. 9.53        3
             Dir       3 d755m      17u    9g                            -1
just_a_file              8        7168         4 15-Aug-86  6.10.24        1
             Data  -5622  644m      17u    9g                          7618
EXAMPLE.UX               1          24         4  6-Nov-86 12.46. 1        1
             Hp-ux -5813  666m      17u    9g                            -1
FILES shown=4 allocated=4 unallocated=18496
BLOCKS (1024 bytes) used=18 unused=21723
```

In the "directory info" column, the first set of characters for the directory TOP, d755m, refer to the mode for the directory. The "d" shows that this file is in fact a directory. The 755 is the octal number defining the access rights (see the Hfs command later in this chapter for a detailed description of access rights and HFS files), and the "m" states that the 755 is the mode value. The 17u, to the right, is the user specification, hence the "u". Finally, to the right of the 17u, is a 9g which is the group specification, denoted by the "g". The numbers 9 and 17 will normally remain unchanged, however, should you need to specify other group or user identifications see the Hfs command in this chapter.

SRM/UX units control access to files using the HFS access right scheme and accordingly this information is printed in the "directory info" column when SRM/UX directories are listed.

Filecopy

The Filecopy command copies a specified file, set of files, or volume to the specified destination.
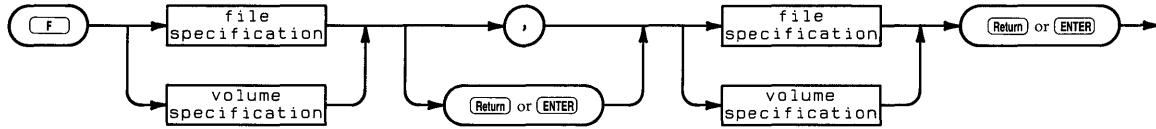
Item	Description	Range
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Filecopy prompt:

```
Filecopy what file ?
```

The Filecopy command is initiated by pressing $\boxed{F}$ and requires two specifications — a source and a destination — separated by either a comma (,) or $\boxed{\text{Return}}$ or $\boxed{\text{Enter}}$. The source volume must be on-line. The destination volume does not have to be on-line (which allows you to copy files on different volumes using a single-volume system).

Copying Single or Multiple Files

To copy files, enter the name of the existing file, followed by the name of the file into which the existing file is to be copied. These two names can be separated either by a comma or by pressing the $\boxed{\text{Return}}$ or $\boxed{\text{Enter}}$ key.

Wildcards may also be used to specify sets of files. If the equals (=) wildcard is used, the copy is not confirmed before taking place. Also, note that if the equals wildcard is used alone (i.e., without any qualifying strings) then the Filer copies every file on the specified volume. If the question mark wildcard is used, you are asked to confirm the copy of each file meeting the wildcard specification before the Filecopy takes place. Thus, using the ? wildcard allows you more flexibility and control over the process.

The dollar sign wildcard ($) may be used in the destination specification to indicate that the file(s) will have the same name (or names) as the source file(s). For example, assume that there are a number of TEXT files on the volume TRIG: and that a second volume named MATH: exists. The following specifications:

```
TRIG:=TEXT,MATH:$
```

result in all the files on the TRIG: volume whose file names end with the string TEXT being copied to the volume MATH: and given the same name as they have on the TRIG: volume.

When copying only files, be sure to use either a file name or the $ character when specifying a destination. If, in the example above, the destination was specified as MATH: instead of MATH:$, the Filer would respond:

```
Destroy EVERYTHING on volume ⟨volume name⟩ ? (Y/N)
```

If you respond with ⎡Y⎤, the directory of that volume will be overwritten. In the case of HFS, the file system may be destroyed. Pressing ⎡N⎤ aborts the Filecopy command and returns the Filer prompt.

Where source and destination are the same volume, the Filecopy command proceeds by reading the first specified file into memory, prompting you to remove that volume and insert the destination volume, and then writing the file in memory onto the destination volume. Depending on the amount of memory in your computer, the amount of material being copied, and the number of files being copied, you may have to swap discs more than once.

Note

When using the Filecopy command with a single mass storage volume, wait for the Filer's prompt before removing the source volume and replacing it with the destination volume. Failure to follow this guideline may result in the loss of information from the source volume.

A size specification may be used in the destination file specification. For example, specifying:

```
SYSVOL:FILE,OTHERVOL:FILE[35]
```

would result in the file being written to the first available area on OTHERVOL: that was at least 35 blocks in size.

Copying Entire Volumes

You can also use the Filecopy command to make a back-up copy of an entire volume. Simply type in the source volume specification and the destination volume specification. The destination volume must have been initialized, but it does not have to have been Zeroed (since the directory gets copied from the source volume). The Filer will ask you if you want the directory destroyed. A volume-to-volume copy makes an **exact** copy of the source volume on the destination volume.

Note that having two volumes with the same name on-line at one time is **not** advised. The Filer looks for volumes according to their volume names and may not be able to distinguish one from the other. Thus, the Filer may perform an action on one volume when you wanted the operation to affect the other volume. The Filer warns you whenever this condition exists. If you get a warning, either remove one of the volumes or use the Filer's Change command to change the name of one of the volumes.

You can copy files on one volume to a volume of a different size, but you should not use volume specifications alone to do this. If the source volume is larger than the destination volume, the Filer refuses to execute the Filecopy. If the source is smaller than the destination, the destination volume ends up the same size as the source when the operation is through, so you lose storage space. (Remember? It makes an exact duplicate of the source.)

The best way to handle copies between different size volumes is to use one of the wildcards. Use the equals wildcard (=) if the destination is larger than the source and the question mark wildcard (?) if the destination is smaller than the source. In the latter case you may have to be selective in your copies, since there may not be enough space for all of the files.

When the Filecopy command has finished its task, the screen displays what file(s) or volume has been copied and the Filer prompt appears. The Filecopy command can be aborted before any specifications are entered by pressing [Return] or [Enter] in response to the prompt.

In cases where the destination volume already contains a file with the same name as the file being copied, this prompt is displayed:

```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

You have the options:

- **Remove:** remove the file before proceeding with the operation.

- **Overwrite:** replace the contents of the old file with the new information. The Overwrite option cannot be used to change the type of a file. Attempting to do so will result in the file contents being inconsistent with the file type.

- **Neither:** cancel the operation, do not copy this file.

The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM and HFS users when duplicate links exist to a file. All links and access rights to the file are accurate when a file is updated with Overwrite, because it is put in the same logical location. If you choose the Remove option, the original file would not actually be removed; only your link to it is removed. The other users are still linked to the original file.

Note

Using the Filecopy command to "copy" a file name to the same name on the same volume results in the file being removed.


Note

The Filecopy command does **not** change the file type. See Translate.

Note

Overwrite of a file of type SYSTM is not recommended, because the start execution address cannot be changed in an existing SYSTM file. Overwrite is also not recommended if the source and destination files are not of the same file type.

Get

The Get command associates a specified file as the current workfile.
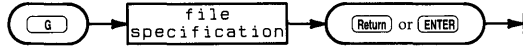
```
( G )──▶│   file    │──▶( (Return) or (ENTER) )──┤
         │specification│
```

Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Get prompt:

```
Get what file ?
```

The Get command is initiated by pressing ⌈G⌋ and prompts you for a file specification. If a workfile currently exists when the Get command is executed, you are asked if you want to release that file before being allowed to specify a new workfile. Upon receiving the specification, the Filer finds the file (or files) and associates that name with the current workfile. Subsequent operations on the workfile use the specified name. The workfile is generally *WORK.TEXT and/or *WORK.CODE.

The Get operation assumes that the text version of the specified file has a .TEXT suffix. If the text version is ASCII, you must include the .ASC suffix, for HP-UX compatible files the .UX suffix. If the text version is Data, you must include a "." at the end of the file name (to prevent the appending of the .TEXT suffix).

The operating system notes that either a text or code or both versions of the workfile exist. Workfiles can only be of type .TEXT/.ASC/.UX/Data or of type CODE. If both text and code versions of the specified file exist, both are associated with the workfile; if only one exists, the association is made with that file. The Filer reports one of three things: either a text or code file has been loaded, both have been loaded, or the file cannot be found on the specified volume.

The Filer is not the only Pascal subsystem where a workfile can be created. The Editor, Compiler, and Assembler subsystems also create workfiles. Once a workfile exists, it is treated as the default file in many of the subsystems. A workfile is "released" by the Filer's New command.

Hfs

The Hfs command permits the specification of access rights to HFS files and directories. (Valid for HFS discs and SRM/UX units only.)
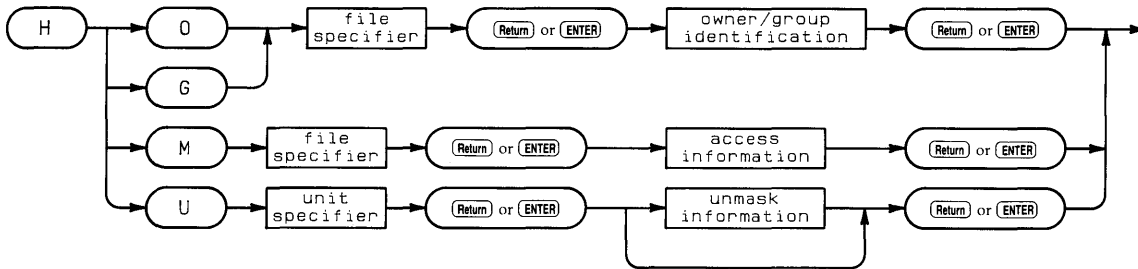
Item	Description	Range
file specification	literal	(See beginning of section)
Owner/Group specification	integer	0 through 65535
Access specification	Access rights entered in octal	0 through 777 (See description of permissible octal codes)
Unit number	integer	(See beginning of section)
Umask specification	integer in octal code	0 through 777 (See description of Umask option)

Semantics

A complete discussion of file ownership is given in the HFS section of Chapter 3. If you have never used HFS discs, you may wish to first read the HFS section.

Caution

Unless you have the HP-UX operating system running on your computer, this command is best left alone. Changing the ownership of a file, when Pascal is the only operating system you have available, is potentially disastrous as all access to the file can be lost!

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, the HP 82901, HP 82902, and HP 9121 drives. 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. The "Adding Modules to INITLIB" section of Chapter 18 discusses the AMIGO and other driver modules.

The Hfs prompt:

```
HFS access: Owner, Group, Mode, Umask, Quit ?
```

is presented to you. Here are your possible courses of action.

- **Owner** allows you to transfer the ownership of a specified file to a different user - providing you are the owner of the file. The required information is the filename and the numeric identification of the new user.

  The Owner option will prompt you:

  ```
  For which file ?
  ```

  Type in the filename, with optional volume specifier and optional directory path. Wild-cards may be used in the filename.

  Once you have typed your filename in and pressed ⌜Return⌝:

  ```
  Enter new owner number
  ```

  The default number for a file created in a Pascal environment is 17. The default number for a file created in a BASIC environment is 18. HP-UX users may also use other numbers, but it should be kept in mind that to transfer ownership of a file from an HP-UX user to a Pascal user, simultaneously transferring all access rights to the Pascal user, requires that the owner number be changed to 17 **while in the HP-UX environment**.

- **Group** permits you to change the group membership of a specified file to some other group - providing you are the owner of the file. Similarly, the required information is the filename and the numeric identification of the new group.

- **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. Information required during this option is the filename and the octal numeric value of the new permissions.

  The default mode assigned to a file when it is created is 666.

  The default mode assigned to a directory when it is created is 777.

  For example, assume that there is a file called TEST.TEXT which has been created with the default access rights of 666 and you now wish to restrict the access of this file. Press ⌜M⌝ to initiate the **Mode** subcommand.

  You will now be prompted:

  ```
  For which file ?
  ```

  Now type the filename which you are operating on. The option will continue with:

  ```
  Enter new mode
  ```

  and respond by entering (for example):

  744 ⌜Return⌝ or ⌜Enter⌝

  This will cause the system to assign read, write and execute rights to you the owner (user number 17), and only read permission to all other users of the system.

This command can be used to grant permissions as well as restrict them, but for either to be possible you must be the owner of the file to be changed.

- **Umask** controls the default access permissions when either a file or directory is first created. Normally new files are created with read and write permissions granted (octal 666), and new directories are created with all permissions granted (octal 777). Using this command you can redefine these default settings by removing the desired permissions which you no longer wish to grant as default. For example, to change the default value for directories to read, write, and execute for the owner *and* execute for the group only, you should enter 067, to obtain the new default permission 710.

  The current value of Umask can be displayed by pressing [Return] when prompted to enter the permissions you want to remove.

  This command will prompt you:

  `For which unit ?`

  You should enter the unit number for which these default values discussed above will be valid. Note that only **one** value can be set per unit.

  You will then be prompted to:

  `Enter new umask number`

  By simply pressing [Return] there will be no change to the umask value for the unit.

For each user class the octal codes are specified as follows:

- If Read permission is to be granted, octal code 4 is specified.

- If Write permission is to be granted, octal code 2 is specified.

- If Execute permission is to be granted, octal code 1 is specified.

The individual permission codes are summed and the final value is the access right for the file for the particular user class. For example, an octal code 664 means that the owner has octal code 6 for the file i.e. read and write permissions granted. The members of the group may also read and write, but other users may only read the file.

If you prefer to consider the code as one number, i.e. six hundred and sixty four in the above example, the following guide will help you to assign the correct value to the file, although the principle used is identical.
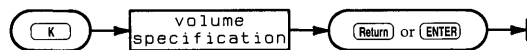
OWNER			GROUP			OTHER		
r	w	x	r	w	x	r	w	x
400	200	100	40	20	10	4	2	1

If you sum all the corresponding numbers where permissions are granted you should obtain the correct octal number. For the above example this would be:

$$400 + 200 + 40 + 20 + 4 = 664$$

Krunch

The Krunch command moves all files on a block structured volume so that all the unused storage space is at the end of the volume.

Item	Description	Range
volume specification	literal	(See the beginning of this section)

Semantics

The Krunch prompt:

```
Crunch what directory ?
```

If there is the slightest question about the reliability of the mass storage medium you are using (because of excessive wear or damage), use the Bad sector command to do a scan of the sectors on the volume **before** initiating Krunch. If a bad sector is found, use the Filer's Make command to make a file of type .BAD over the bad sectors. Krunch does not move files of type .BAD. Moving files onto an unreliable area of storage is a good way to lose them.

The Krunch command is initiated by pressing ⬚K⬚ and it prompts you for a volume specification. After you respond with a legal volume specification of an on-line, block-structured volume, it prompts:

```
Crunch directory MKWORK: ? (Y/N)
```

Where MKWORK: is whatever volume you specified. Typing ⬚Y⬚ for Yes lets the command continue; ⬚N⬚ for No returns the Filer prompt. The Krunch command executes a sensitive operation — that of moving all the files forward on the disc by reading the files into memory and then writing them back out on the disc in such a manner so as to make all the unused space on the volume contiguous at the end of the disc.

Caution

UNDER NO CIRCUMSTANCES SHOULD YOU ATTEMPT TO IN-TERRUPT THE KRUNCH OPERATION ONCE IT HAS BEGUN. You are risking your directory, and thus all of the information contained on that medium if you do so. Do not touch the power switch or the door on the disc drive, or attempt to use the keyboard while a Krunch is in progress.


This process becomes necessary when, after repetitive reading and writing to the disc, the available storage space becomes highly fragmented. For instance, suppose you have 100 blocks available on the disc, but because they are all in 10 or 15 block chunks, there is not enough contiguous storage space for the system to write a 20 block file to the disc.

The Krunch command is extremely useful and using it should not worry you. However, because it alters the directory (which maps where the information on the disc resides), it is one of the quickest ways to wipe out a volume. The precautions outlined above should help you avoid any problems while using the command.

The Krunch command does nothing on SRM, SRM/UX, or HFS units.

List directory

The List directory command lists directory information about a block-structured volume or one of its subsets.
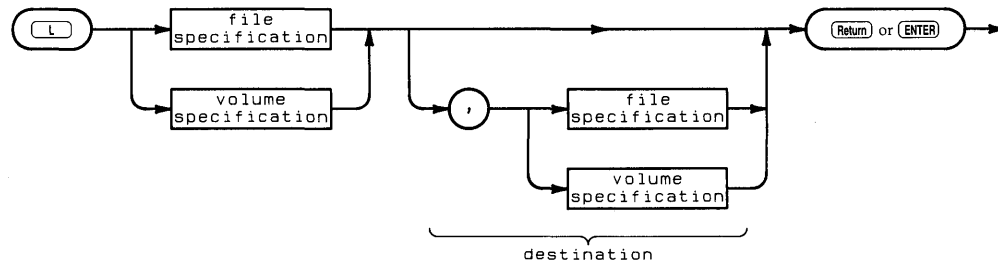
Item	Description	Range
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The List directory prompt:

```
List what directory ?
```

The List directory command requires a legal volume or file specification. Results can be listed to the PRINTER: or to a text file if specified and separated from the first specification by a comma. If no destination is specified the listing defaults to the CONSOLE. Wildcards are available to identify subsets of files on the volume.

In the listing, the name of the volume is displayed in the upper left-hand corner. To the right, the directory type is displayed. Pascal LIF discs have Level 1 directories. Level 1 directories contain directory-create and volume size information. Level 0 directories (created on other systems) do not. If your directory is of type LIF, it should display the date the directory was created and the date it was changed as system volume, the size of the storage blocks, and whether the listing is in storage order or alphabetical order. The size of blocks on LIF volumes is 256 bytes. The size of blocks on WS1.0 volumes is 512 bytes. The Shared Resource Management system uses single-byte "blocks", and HFS discs usually use blocks of size 1024 bytes.

To have directories listed in alphabetical order, include [*] after the directory name. For example:

```
MYDIR:[*]
```

The column entries for each file include: file name, number of blocks used for storage, the file size in bytes, and the date the file was created or changed.

The last two lines display additional directory information. On LIF volumes this includes how many more entries can fit in the directory. The size of a directory is specified when the disc is initialized. You need one 256 byte block for each eight directory entries.

For example, initiating the command by pressing ⎣L⎦, specifying **ACCESS**: and pressing ⎣Return⎦ or
⎣Enter⎦ results in the following listing appearing on the screen:

```
ACCESS:              Directory type= LIF level 1
created 20-Sep-82 13.57.17 block size=256
 Storage order
...file name....     # blks    # bytes  last chng

FILER                  218        55808 20-Sep-82
EDITOR                 224        57344 20-Sep-82
LIBRARIAN              202        51712 20-Sep-82
MEDIAINIT.CODE         132        33792 20-Sep-82
TAPEBKUP.CODE           54        13824 20-Sep-82
FILES shown=5 allocated=5 unallocated=3
BLOCKS (256 bytes) used=830 unused=223 largest space=223
```

The Extended Directory command gives more information about the files and unused areas on
the volume.

Make

The Make command creates files and directories.
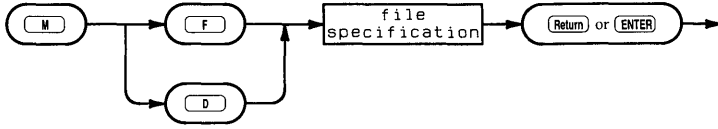
Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Make prompt:

```
Make File or Directory ? (F/D)
```

The Make command is useful primarily in two ways. Files can be made when you need to reserve physical space on a disc, and directories can be made on an SRM or SRM/UX system, or an HFS disc.

Making Files

The Make command is **not** required to create files to be used by the various Pascal subsystems. It reserves space only; it in no way initializes or changes the contents of the space. In the Pascal System, each subsystem lets you either create or specify any files you need. Users of HP BASIC may quite naturally think that the same function is served by this command as the CREATE command in BASIC (where you must create a file before using it). Thus the distinction between these similar sounding commands is drawn here.

The Make command requires at least a file specification (which includes a volume specification by definition) and accepts an optional size specification. If the (positive integer) size is given, it must follow the file specification on the same line and be enclosed in square brackets. The Filer then creates a file with the specified name and of the specified size on the first area of the volume that has a large enough area of contiguous storage space to meet the size requirements.

When using a size specification to make a file, you must be aware that the size is specified in "number of blocks". The size of all "Make" blocks is 512 bytes — regardless of the directory type. A LIF directory considers a 256 byte sector to be a block. The WS1.0 directory considers a block to be 512 bytes. So if you make a file on a LIF volume and specify 500 blocks, it will show up in the directory as 1000 blocks.

For example, assume that there is a volume named MKWORK: on-line that has at least 22 blocks of contiguous and unused space available. Press ⬚M⬚ to initiate Make, specifying:

```
MKWORK:DUX.TEXT[22]    Return  or  Enter
```

This results in a file named DUX.TEXT being created on the first available area with 22 blocks of the volume MKWORK: and the Filer reporting the following:

```
MKWORK:DUX.TEXT made
```

A subsequent listing of the directory (using the List directory or Extended directory commands) will show a file of the same name with a 22-block size (on WS1.0 directories).

The size specification may be omitted, in which case the Filer creates the specified file using the largest unused area on the disc (i.e., the largest contiguous storage space on the disc will be allocated to the file). It is recommended that you specify the size you want the file to be.

There are two special cases of size specification worth knowing about. The first is the number zero enclosed in brackets [0] which is the same as omitting the size specification altogether — the Filer uses the largest space available. The second case is the asterisk character enclosed in brackets [*] which tells the Filer to make the file's size either the second largest area on the disc or half of the largest area, whichever is greater. These two special cases are ignored by HFS discs, and SRM systems will ensure there is enough space available and make a zero size file.

Rebuilding Files

The Make command is useful if you must rebuild a file that was lost on a LIF disc. Here are steps you can take to do that.

1. You must know its size and where it was located.

2. Then make TEMP files (TEMP1, TEMP2 etc.) over all the unused spaces on the disc that are as large or larger than the file you'll be making.

3. Then make a file of the proper type over the lost file to recover it.

4. Finally, use the Filer's Remove command to remove all the TEMP files.

An Extended directory listing can help you determine the location and size of unused areas on the disc.

The above techniques will not recapture lost files on SRM, SRM/UX, or HFS systems.

Making Directories (SRM, SRM/UX, or HFS)

The Make command is used to create directores on an SRM or SRM/UX system or HFS disc. For example:

Answer the first question by typing ⌷D⌷ and specify where you want the directory located and what is its name. The directory path tells where you want it and its name is the name on the end of the path. For example, if you had a directory:

```
/USERS/JOE/PROJECT1
```

If you wanted to create a directory for Project 1's DATA files, you should type:

```
/USERS/JOE/PROJECT1/DATA
```

The DATA directory is created in the PROJECT1 directory.

New

The New command releases or clears the workfile(s).

Semantics

The New command requires no specifications. Upon pressing $\boxed{\text{N}}$ to initiate the command, it clears the workfile unless the workfile has been updated since the last Save command. If the workfile has been updated but not Saved, the following prompt appears:

Throw away current workfile? (Y/N)

Responding by pressing $\boxed{\text{N}}$ for No allows you to use the Save command to write the file to a volume; $\boxed{\text{Y}}$ for Yes clears the current workfile area.

You can check the status of the workfile before using New with the What command. The What command gives you the name and status (saved or not) of the current workfile.

After the Filer executes the New command, it will respond with:

Workfile cleared

Do not confuse the Filer's New command with the New system volume command at the Main Command Level — the two commands are different and perform separate functions.

Prefix

The Prefix command changes the default volume to the one specified.
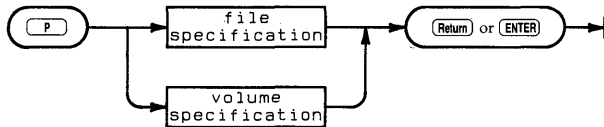
Item	Description	Range
file specification	literal	for SRM, SRM/UX, and HFS only (See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Prefix prompt:

    Prefix to what directory ?

The Prefix command is initiated by pressing ⌐P⌐ and requires a volume specification. The command allows you to specify a new default volume — the volume where the Filer searches for file names when a volume name is not specified. The volume must be block structured (one used for mass storage) but does not have to be on-line. The current prefix (i.e., default) volume can be obtained by responding to the Prefix prompt with a colon (:) or just ⌐Return⌐ or ⌐Enter⌐. (The Volumes command may also be used).

When the command executes, the screen displays the message:

    Prefix is MKWORK:

where MKWORK: is the name of the current default prefix. The Prefix command saves keystrokes if you are doing a lot of file accessing on a particular volume.

Filer commands which request a volume specification may be answered with the colon character (:) which specifies the current default volume.

It is possible to set the default prefix to a flexible disc drive, regardless of the volume inside. This is done by typing the following while the drive door is open or the drive is empty.

    #3: ⌐Return⌐ or ⌐Enter⌐

The prefix command is used to set up a working directory on an HFS disc or SRM or SRM/UX system. If you had an SRM file named:

    #5:USERS/JOE/PROJECT1/PROGRAMS/FILE

Initiate the Prefix command as usual and specify:

```
#5:USERS/JOE/PROJECT1/PROGRAMS
```

This sets the SRM volume as the default volume (with volume name of PROGRAMS) and USERS/JOE/PROJECT1/PROGRAMS as the working directory on the SRM. Now you can specify the file with:

```
FILE
```

If you were to use the Prefix command again to set the default prefix to another volume (not on the same unit), the working directory and volume name for the unit remain PROGRAMS. You need only specify:

```
#5:FILE
```
or
```
PROGRAMS:FILE
```

Either will get the same file. The principle is exactly the same for HFS.

Note

Do **not** use the Prefix command on unit #45 of #46. These are the system volumes for SRM and HFS, respectively, and should not be altered.

When Prefixing an HFS-formatted **flexible** disc, you are automatically prefixed to the root directory; you **cannot** prefix to any directory below the root level on this type of flexible disc. (However, with HFS-formatted **hard** discs, this restriction does **not** apply.)

It is possible to change the working directory on an SRM unit without changing the default volume. Use the Filer's Unit directory command. Press ⎡U⎤ and then give the directory name that you wish to become the working directory. If the new directory is in the existing working directory, just type the new directory name. If it is not, type the whole directory path as shown above in the Prefix example.

Quit

The Quit command exits the Filer subsystem and returns control to the Main Command Level.

```
( Q )—►
```
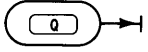
Semantics

The Quit command has no parameters and no specifications of any type are needed. Pressing `Q` exits you from the Filer and the Main Command Prompt is displayed on the screen.

Remove

The Remove command purges specified files from the directory.
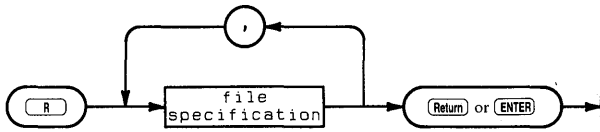
Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Remove prompt:

```
Remove what file ?
```

The Remove command is initiated by pressing ⎡R⎤ and requires a file specification. The command removes the specified file from the directory, updates the directory, and reports the action it has performed. Wildcards may be used to specify a subset of files to be removed. If the equals wildcard (=) is used in the file specification, the Filer reports the specified file or files and then prompts:

```
Proceed with remove ? (Y/N)
```

This is the last chance you have to change your mind about the removal. Pressing ⎡N⎤ for No aborts the operation and no files are removed. Pressing ⎡Y⎤ for Yes removes those files meeting the wildcard specification from the directory. The process is not always reversible. However, the Make command can sometimes be used to recover a removed file on a LIF or WS1.0 volume.

Note

The Filer considers the file specification = to specify ALL the files on the default volume and MKWORK:= to specify ALL the files on the MKWORK: volume. If you use the wildcard in this form and respond to the Filer's prompt (**Proceed with remove ? (Y/N)**) with a ⎡Y⎤ for Yes, every file on the directory of the specified volume is removed. Responding with a ⎡N⎤ for No aborts the operation. Wildcards can be hazardous to your files — watch the prompts.

Specifying a single file (of an on-line volume of course) in response to the Remove prompt results in the removal of that file from the directory and a report that the file has been removed. Once the ⌈Return⌉ or ⌈Enter⌉ key is pressed following the file specification (unless wildcards are used), that file is gone.

While the use of the equals wildcard (=) results in being prompted for whether or not you want the directory updated, the question mark wildcard (?) acts slightly differently. It allows you to be more selective in your removal. Given the volume PROCESS: containing the files:

```
NOVMEMO.TEXT
MARKLTR.TEXT
PARSER.TEXT
PARSER.CODE
GARBAGE.TEXT
```

The specification **PROCESS:?TEXT** in response to the Remove prompt results in the screen clearing and the following message appearing.

```
Remove NOVMEMO.TEXT ? (Y/N)
```

Answering with either a ⌈Y⌉ or ⌈N⌉ results in the next prompt appearing below the first:

```
Remove MARKLTR.TEXT ? (Y/N)
```

The process continues until you have been prompted for all the TEXT files on the **PROCESS:** volume and then the final prompt appears:

```
Proceed with remove ? (Y/N)
```

You may be respond with either a ⌈Y⌉ (for Yes) or ⌈N⌉ (for No). The files are not actually removed until this prompt is answered with a ⌈Y⌉. The **?** wildcard thus allows you to be both selective and relatively safe about your file removals.

The Remove operation treats HFS and SRM directories like files if they are empty. Remove is not allowed on non-empty HFS and SRM directories.

Save

The Save command saves the current workfile on the specified volume.
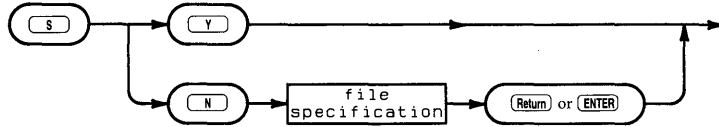
Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Save command is initiated by pressing ⎡S⎤ and may or may not require a file specification.

If the workfile was previously named using the Save command, or originally obtained using the Get command, then the Filer prompts:

```
Save as PREVIOUS.TEXT ? (Y/N)
```

Where `PREVIOUS.TEXT` is the name previously associated with the workfile. Responding with a ⎡Y⎤ for Yes results in either a CODE or TEXT file (or both, depending on what is in the workfile) of that name being removed and replaced with the current workfile.

If the workfile was never updated, it is automatically saved with the original name.

If the workfile is not named, or if you answer ⎡N⎤, the Filer prompts:

```
Save as what file ?
```

When naming the file, the following conventions apply to the type of the file:

1. If a standard suffix is recognized, the workfile is either Filecopied, Translated, or Changed (on the system volume) to the file name and type.

2. If no suffix is recognized, a .TEXT file is the default.

3. If no suffix is included, but a trailing period (.) is found, the file type is Data.

4. The .CODE file is created by removing the suffix (if there is one) and adding .CODE to the file name.

The Filer displays that the file is now saved.

To find out what the current name and state (saved or not) of the workfile is, use the Filer's What command.

Translate

The Translate command converts text files among the TEXT, ASCII, UX, and DATA types.
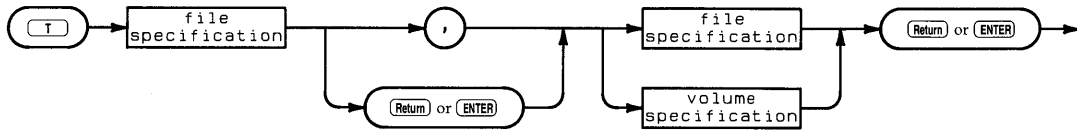
Item	Description	Range
file specification	literal	(See the beginning of this section)
volume specification	literal	(See the beginning of this section)

Semantics

The Translate prompt:

```
Translate what file ?
```

The Translate command is initiated by pressing [T] and requires two specifications — a source and a destination separated by either a comma (,) or a carriage return (press [Return] or [Enter]). The source specification can be any block structured volume, any file, or any group of files on a volume. The destination specified can be any of the above and may also be a non-block structured volume (i.e., the PRINTER: or CONSOLE:). Non-block structured volumes (like the PRINTER:) are assumed to be on-line.

Wildcards may be used to specify sets of files but if a wildcard is used in the source specification, either a wildcard or the $ character (discussed below) must be included for the destination. If the equals wildcard (=) is used, the Translate is not confirmed before taking place. Also, note that if the = wildcard is used alone (i.e., without any qualifying strings such as TEXT, CODE, etc.) then the Filer Translates every file on the specified volume. If the question mark wildcard (?) is used, you are asked to verify the translate of each file meeting the wildcard specification before the Translate takes place. Thus, using the ? wildcard allows you more flexibility and control over the process.

The dollar sign wildcard ($) may be used in the destination specification to indicate that the file(s) will have the same name (or names) as the source file(s). For example, assuming that there are a number of TEXT files on the volume TRIG: and that a second volume named MATH: exists,

```
TRIG:=TEXT,MATH:$
```

This results in all the files on the TRIG: volume whose file names end with the string TEXT being translated to the volume MATH: and given the same name as they have on the TRIG: volume.

When source and destination files are on the same volume, the Translate command proceeds by reading the first specified file into memory, prompting you to remove that volume and insert the destination volume, and then writing the file in memory to the destination volume. Depending on the amount of memory in your computer, the amount of material being translated, and the number of files being translated, you may have to swap discs more than once.

Note

When using the Translate command with a single-volume, wait for the Filer's prompt before removing the source volume and replacing it with the destination volume. Failure to follow this guideline may result in the loss of information from the source volume.

The Translate command allows the translating of files or groups of files to non-block structured devices like the **PRINTER**: and **CONSOLE**:. Only text files (i.e., of type TEXT, ASCII, UX, or Data) should be sent to printers since other files are not generally human-readable.

When the Translate command has finished its task, the screen displays what file(s) have been translated and the Filer prompt appears. The Translate command can be aborted before all specifications are given by pressing [Clear line] ([Clear line]), then [Return] or [Enter].

In cases where the destination volume already contains a file with the same name as the file being Translated, this prompt is displayed:

```
ANYVOL:XFILE
exists ... Remove, Overwrite, Neither ? (R/O/N)
```

You have the options:

- **Remove:** remove the existing file before proceeding with the translation.

- **Overwrite:** replace the contents of the old file with the new information. The Overwrite option cannot be used to change the type of a file.

- **Neither:** cancel the operation.

The Overwrite option allows you to put a file in the same starting location as the original. This is important to SRM, SRM/UX and HFS users when duplicate links, passwords, etc. exist to a file. All links and access rights to the file are accurate when a file is updated because it is put in the same logical location. If you chose the Remove option, the original file would not actually be removed; only your link to it is removed. The other directories are still linked to the original file.

The Translate command can be used to send files to an SRM printer by copying the file into a special *spooler directory.*

For example, to print the text file named `JOB_1.TEXT` located by the directory path `#5:/PROJECT_1` on the printer assigned to spooler directory named `LP`:

1. Type `F` to enter the Filer from the Main Command Level.

2. Type `T` to invoke the Translate command.

   The Filer prompts:

   Translate what file?

3. Type:

   `#5:/PROJECT_1/JOB_1.TEXT,#5:/LP/JOB_1.ASC` `Enter` or `Return`

   The file will be printed as soon as the printer is available. The `.ASC` ending on the file name tells the Filer to translate the information into ASCII format, which is the format handled by the SRM spooler and its supported peripherals.

Unit directory

The Unit directory command changes the volume name and working directory for an SRM or HFS unit.
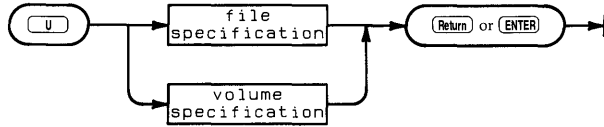
Item	Description	Range
file specification	literal	(See the beginning of this section)

Semantics

The Unit directory prompt:

```
Set unit to what directory ?
```

The Unit command changes the working directory on HFS, SRM, and SRM/UX units. The working diretory and the volume name for HFS, SRM, or SRM/UX units are the same. The Prefix command performs the same operation but sets the default volume to the HFS, SRM, or SRM/UX volume. The Unit command does not.

To specify the working directory, you must start either from the existing working directory or from the root directory. To get to the root directory, supply the volume name or unit number for an HFS, SRM, or SRM/UX unit, followed by / (e.g., #5:/). This positions you in the root directory. From the working directory, you can continue down the tree structure from directory to directory, or you can go back up the structure one directory at a time using ".." for the parent of the current directory. For example:

If the present working directory for unit #5 is:	USERS/JOE/PROJECT1/PROGRAMS
and you wanted the new working directory to be:	USERS/JOE/PROJECT5/DOCUMENTS
you can specify it in one of these ways:	#5:/USERS/JOE/PROJECT5/DOCUMENTS or PROGRAMS:/USERS/JOE/PROJECT5/DOCUMENTS or #5:../../PROJECT5/DOCUMENTS or PROGRAMS:../../PROJECT5/DOCUMENTS

Note

You cannot set the working directory for an HFS-formatted flexible disk to any directory below the root directory; the Filer will display an error message if you attempt this. This restriction does not apply to HFS hard discs.

Volumes

The Volumes command lists the volumes currently on-line.

Semantics

The Volumes command requires no specifications. Upon pressing $\boxed{V}$ it displays the following information about all on-line volumes currently associated with the Pascal Workstation File System: the logical unit number associated with a volume; whether the volume is the system (boot) volume, a block-structured volume or a non-block-structured volume; the volume's name; and the current Prefix or default volume.

This is a typical display generated by the Volumes command:

```
Volumes on-line:
   1   CONSOLE:
   2   SYSTERM:
   3 # MINI3:
   4 # MINI4:
   5 # MY_SRM:
   6   PRINTER:
  45 * SYSTEMO4:
Prefix is - MY_SRM:
```

The number on the far left is the logical unit number associated with the volume. The * character in the second column indicates the system volume, which is always block structured. The # character indicates all other block structured volumes currently on-line. The remaining volumes (shown with no character in the second column) are non-block-structured. The last line of the display shows the current default volume. It is where the system looks for a file when no volume has been specified.

The above configuration shows two flexible disc drives associated with units #3 and #4, and two SRM volumes associated with units #5 and #45; the SRM units are the working volume and system volume, respectively.

What

The What command displays the name and state (saved or not) of the workfile.

```
( W )—▶—|
```

Semantics

The What command is initiated by pressing `W` and requires no other input. The command shows the name of the current workfile or indicates that it is not associated with a file name. It also shows whether or not the workfile has been Saved since the last update to the file. If no workfile exists, the Filer responds with:

`No workfile`

Suppose you had two files named `INFRARED.TEXT` and `INFRARED.CODE` on the default or prefix volume. Assume that you used the Filer's Get command and specified `INFRARED` to associate the files with the workfile. If you then edited the TEXT version of that file (using the Pascal Editor), returned to the Filer and executed the What command, the screen would display:

`Workfile is INFRARED (not saved)`

because the workfile was changed since the last time a Save command was executed.

Saving the workfile does not change the fact that the workfile exists. It is still there. The New command is used to clear the workfile.

Saving the workfile is not remembered between separate sessions of the Filer. If you Save the workfile during the current Filer session, a New command immediately clears the workfile. If you Save it, quit the Filer and then return to use the New command, the Filer will ask:

`Throw away current workfile ? (Y/N)`

even though you saved it during the previous Filer session and haven't updated it since.

Zero

The Zero command creates an empty directory on the specified volume. (The Zero command is not allowed on SRM or SRM/UX volumes or HFS discs. Use the Make command or the MKHFS utility.)

```
( Z )──▶[ volume
         specification]──▶( (Return) or (ENTER) )──▶|
```

Item	Description	Range
volume specification	literal	(See the beginning of this section)

Semantics

The Zero prompt:

```
Zero directory (NOT valid on HFS and SRM type units)
Zero what directory ?
```

The Zero command is initiated by pressing [Z] and requires the volume specification of a block-structured volume. The volume must be formatted using the Pascal utility program named MEDIAINIT.CODE supplied on the ACCESS: volume.

Since the Zero command creates an new empty directory on the volume, you will be prompted:

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

Responding with a [N] for No aborts the command and returns the Filer prompt.

If you answer [Y], the next prompt is:

```
Number of directory entries (80) ?
```

The number in the parentheses is the number in the existing directory. Respond with [Return] or [Enter] if that is the number you want. If there is no number in parentheses, [Return] or [Enter] causes the default number for that directory type (80 for LIF; 77 for WS1.0) to be put on the disc.

The next prompt is:

```
Number of bytes (270336) ?
```

It is asking for the logical size of the disc (the extent to be managed by the directory). The number in the parenthesis is the number in the existing directory or the default for that disc. Press [Return] or [Enter] to use the displayed number.

The next prompt is:

```
New volume name ?
```

The Filer is asking for a legal volume name. Volume name formats vary with different directory structures. LIF directories allow up to six characters with uppercase and lowercase characters being distinct. WS1.0 directories allow up to seven characters, and all are made uppercase.

An answer of [Return] or [Enter] aborts the Zero command.

After typing a volume name, the final prompt appears:

    NEWSTUF: correct ?

Responding with [N] aborts the Zero command. Responding with [Y] results in the message:

    NEWSTUF: zeroed

Where NEWSTUF: is the name of the new volume. The Filer prompt reappears when the operation is complete.

Note

Because the Pascal File System works with volume names, a LIF volume whose name is all blanks (ASCII spaces) will not be recognized as a valid volume.

Pascal Compiler

6

Introduction

This chapter describes the Workstation Pascal Compiler, another subsystem of the Workstation Pascal System. It shows how to use the Compiler to prepare Pascal source programs for execution on the Workstation System.

The Workstation Pascal Compiler supports a generous set of Pascal language features. They are briefly described in the "Overview of Workstation Software Features" chapter in Volume 2. Further details of the language features available on this system are referenced in that chapter.

Two slightly different compilers are shipped with the Workstation System:

- COMPILER produces object code for all the processors in the MC68000 processor family (because it generates only MC68000 instructions).

- COMPILE20 produces machine-specific object code for the MC68020 and MC68030 processors (the MC68000 and MC68010 processors cannot execute these instructions).

Preparing a program for execution on this system is a simple process. First, produce source text file(s), usually with the Editor subsystem. Then use one of the two compiler subsystems to generate an output file of relocatable object code. This output file is ready to be linked and run with the Run command — normally there is no explicit link step.

Compilation speed depends on the storage medium where the source and object code reside. Using floppy discs, about 1600 lines per minute (lpm) is typical. If the files are memory-resident, the rate is around 4000 lpm. The Compiler's speed contributes significantly to the interactive and crisp feeling of the Workstation Pascal environment.

The Compiler, supported by other subsystems, provides complete facilities for the creation, maintenance and use of software libraries. Modules of Pascal code can be compiled, stored in the System Library, and automatically accessed by any program which needs them. Compiled modules carry along a detailed specification of their interface which allows any other program or module to use the code or data structures they declare.

Object code produced by the Compiler (and the Assembler and Librarian, for that matter) is targeted specifically for 3.1 and 3.2 versions of the Pascal Workstation. It is in the Workstation Loader format, and is not generally suitable for loading and execution on other operating systems. A notable exception is the capability to make CSUBS (compiled subprograms) for BASIC with the Compiler or Assembler, and the CSUB utility which is a separate product that executes on the Workstation.

Steps In Program Development

This section will teach you by example the steps required to compile and run a simple program. You need to know how to use the Editor before you can proceed with this material. We begin at the Main Command Level of the system, with no workfile present.

Prepare the Source Program

First we need a program to compile. Enter the following program using the Editor. The Compiler isn't particular about margins, so you can adjust the program to the left margin as you type. Try to preserve the indentation, to keep the program easily readable by mortals.

Notice that the word "end" is intentionally misspelled at the bottom of the program. Type it just as shown, so you can see how errors are handled.

When you leave the Editor (Quit command), you should specify that the output is to be written to the file "HOWDY". Don't make a workfile (don't use the Update option).

```
program howdy (input,output);
type
  color = (red,orange,yellow,green);
var
  hue: color;
  i: integer;

  procedure show (c:color);
  begin
    writeln(output,'Howdy!  ',c);
    i := i+1;
  end;

begin
  writeln(output);  i := 0;
  for hue := red to green do
    show(hue);
emd.
```

At this point, if you use the Filer to examine the directory of your default volume, you'll see the file "HOWDY.TEXT" .

Invoke the Compiler

The Compiler is invoked by typing the `C` key when the system is at the Main Command Level. At the time you booted up, the system looked for the Compiler on all the mass storage volumes which were on-line. If the Compiler was found at that time, it is expected to still be in the same volume whenever it's needed. If the Compiler wasn't found, the system will try to run CMP:COMPILER (or the file specified with the last What command).

So if you press `C` and the system responds that it can't load the Compiler, you must **first** put the CMP: disc in a drive , **then** press `C` again.

It takes a few seconds for the Compiler to load from a floppy disc. Then it will ask you:

    Compile what text ?

If you had to swap discs, you should remove the CMP: disc and put back the default volume. Now respond:

    HOWDY [Return] or [Enter]

The Compiler automatically appends the ".TEXT" suffix to the name you give; you need not do so yourself. Next you are asked:

    Printer listing (l/y/n/e) ?

If you have no printer, you **must** answer `N` for no listing, or `L` for a listing file. If you've got a printer, the `Y` response gets you a complete listing. Answering `E` will get you a listing only of any errors which are detected. For the moment, let's answer `N` and get no listing. Finally the compiler asks:

    Output file (default is "HOWDY.CODE") ?

Respond to this by pressing [Return] or [Enter] to accept the default.

As the Compiler runs, you can observe its progress through the source program. Each dot displayed represents five lines of the source text which have been scanned. Whenever the body (the "begin") of a new procedure is reached, that procedure's name is displayed on the screen along with an estimate (in square brackets) of how much memory is still available for the Compiler to use. The Compiler reads through an entire procedure body before generating any code; if you write very large procedures, you may notice the stream of dots hesitating momentarily at the ends of some of them.

When the misspelled word "emd" is encountered, the Compiler will beep and display the offending line. You now have three options: press the space bar to continue compiling; hold down [Shift] and press [Select] ([EXECUTE]) to terminate the compilation; or enter the Editor to fix the mistake. In this example, you should select "Edit" by pressing `E`.

Note

The Editor must be Permanently loaded, or the volume containing the Editor must be on-line to use the `E` option when exiting the Compiler.

Handling Syntax Errors

When compiling, errors are printed on the screen and the Compiler pauses to ask if you wish to edit your file. If you specify a compiler listing file, it will contain all error messages. In this case, you must call the Editor yourself after the compilation is finished.

When the Compiler points out a syntax error, the place it indicates is not necessarily the place where the error occurred; rather, you are shown where the error was first **recognized**. An easy way to get extreme examples of this is to accidentally have unbalanced "begin" and "end" pairs in a deeply nested program. The imbalance may be **syntactically** (though not visually) undetectable until much later in the program. Compilers don't see what you mean, only what you write.

The error message may not seem reasonable to you. For instance, your misspelled "end" looks to the Compiler like an undeclared identifier which may be the beginning of an assignment statement. The Compiler sees no similarity between "end" and "emd".

When an error is detected, the Compiler tries to recover by making an assumption about what you meant. Frequently the assumption is wrong, which leads to further errors being reported in the vicinity of the first one. Sometimes the Compiler will try to recover by skipping text until it sees a keyword or other symbol it recognizes.

Back to the example: you elected to edit the program, so the Compiler terminated and the Editor is now invoked. The file containing the offending line is automatically brought in, and the cursor is placed where the error was reported. Simply fix the misspelling and quit the Editor, using the Save option to rewrite the corrected file under its original name, "HOWDY".

Repeat the steps above to compile HOWDY again. If you have a printer, this time you should ask for a listing. If there are no other accidental errors, the compilation will succeed this time. Your printout should look like this:

```
Pascal [Rev 3.2  1/15/87] HOWDY.TEXT              19-Feb-87 15:28:20 Page 1

      1:D         0 program howdy (input, output);
      2:D         1 type
      3:D         1   color = (red,orange,yellow,green);
      4:D         1 var
      5:D     -2  1   hue : color;
      6:D     -6  1   i : integer;
      7:S
      8:D         1   procedure show (c:color);
      9:C         2   begin
     10:C         2     writeln(output,'Howdy!  ',c);
     11:C         2     i := i+1;
     12:C         2   end;
     13:S
     14:C         1 begin
     15:C         1   writeln(output); i := 0;
     16:C         1   for hue := red to green do
     17:C         2      show(hue);
     18:C         1 end.

No errors. No warnings.
```

Interpreting the Compilation Listing

The column of numbers at the left enumerates the lines. "D" next to the line number indicates the line is a declaration; "S" indicates the line was skipped altogether, either because it's blank, or because it is entirely within a comment. "C" indicates the line is part of the body of a Pascal block.

The two numbers, −2 and −6, provide information about where the variables "hue" and "color" will be stored in memory. More detailed information about this can be requested by the **$TABLES$** Compiler option.

The column of numbers immediately to the left of the program text shows how deep structures in the program are nested. This can be very useful when **begin** and **end** statements get out of balance. The main program is at level 1, with procedures nesting successively deeper. The structural nesting of complex statements such as **FOR**-loops, and **IF** and **WITH** statements is also counted.

Running the Compiled Program

If you use the Filer to look at the directory of your default volume, you'll see that there are two HOWDY files now: HOWDY.TEXT and HOWDY.CODE . Press the ⌐R⌐ or **RUN** key. The operating system remembers the name of the most recently compiled file. You'll see the message:

```
Loading 'HOWDY.CODE'
```

The program runs, producing this display on the screen:

```
Command: Compiler Editor Filer Initialize Librarian Run eXecute Version ?

Howdy!  RED
Howdy!  ORANGE
Howdy!  YELLOW
Howdy!  GREEN
```

You can also run the program by using the eXecute command: press ⌐X⌐ or the [Select] ([EXECUTE]) key. Then when asked

```
Execute what file ?
```
answer
```
HOWDY  Return or [ENTER]
```

Try it now. Actually, you can eXecute any program, not just the one you most recently compiled. Also, if you use the Run command when you haven't compiled any program, the behavior is as if you used the eXecute command.

Using a Workfile

The Compiler's behavior depends somewhat on whether you are compiling a workfile, or some other source file. If you use a workfile, you are asked fewer questions by the Compiler and Editor; in fact, while the workfile is present you can't compile or edit any other file! This kind of abbreviated behavior may be a blessing or a curse, depending on your needs.

Workfiles are most useful when you're writing, compiling, and iteratively refining a single program source. In such a case, you'll appreciate the convenient reduction in keystrokes needed to edit, compile, and run the program over and over. On the other hand, experienced programmers developing complex systems with many source files almost never use workfiles.

Workfiles are **not** particularly useful unless the Editor has been permanently loaded, or the volume containing the Editor is on-line.

There are two ways to tell the system to use a workfile. You can create one by using the Update option when quitting the Editor; a workfile made this way will always be called WORK.TEXT, and it will be stored on the system volume. Alternatively, you can designate some existing file as the workfile by using the Filer's Get command. The Update option and the Get command are explained in the Editor and Filer sections of this manual.

Let's make a workfile of HOWDY using the Editor. Press the ⎡E⎦ key, and answer that you want to edit HOWDY. Immediately use the Quit command, and select the option to Update the workfile. This makes a **copy** of your original source file (but not of the code file). Note that the system volume must be on-line at this point, since that is where the workfile is kept.

Now press ⎡R⎦. If the Compiler isn't on-line, you will need to insert your CMP: disc first. If you swapped discs, then after the Compiler is loaded it will say:

```
Mount *WORK.TEXT and press <space>
```

As you can see, the Compiler knows it's supposed to compile the workfile, and you must put your system volume back in the drive. If the Compiler was already on-line, only one question is asked after you press ⎡R⎦:

```
Printer listing (l/y/n/e) ?
```

Probably you'll answer no. The program is then compiled, producing WORK.CODE, and immediately run.

To execute it again, just press ⎡R⎦. It won't be recompiled unless you change it with the Editor. If you aren't convinced it actually ran again (it happens pretty fast), press the space bar to clear the screen before running it again.

Debugging

The Debugger subsystem is described in detail in Chapter 9 of this manual.

Note

With Pascal 3.0 and later versions, the Debugger is not automatically loaded at boot time. You will need to load it if you want to use it. See the Debugger chapter for loading instructions.

Modules

A **module** is a program fragment which can be compiled independently and later used to complete otherwise incomplete programs. For example, you might want to define a "complex number" data type and some relevant functions, then use those definitions in several programs. This section introduces the concepts and facilities you will need to define, debug, and use module libraries.

Modules, like almost everything else in Pascal, must have all their relevant features and characteristics declared before use. Diagrams precisely detailing the syntax of a module declaration can be found in the *HP Pascal Language Reference* manual; an information presentation is more suitable for present purposes.

Module Structure

The four parts of a module are its heading, the import and export sections, and its implement part.

- The **heading** introduces the module and names it. The name is an ordinary Pascal identifier. Example:

  ```
  module complexmath;
  ```

- The **import** part names all other modules on which the present one depends. One module depends on another if the dependent module makes use of things exported from the imported one: calling procedures, assigning to exported variables, or declaring variables of an exported type. The names are separated by commas and the list ends with a semicolon:

  ```
  import complexmath,conversions;
  ```

  There is no import part if the module is independent of all others.

- The **export** part defines the constants, types, variables, procedures and functions which this module will supply to any program or module importing it. Constants, types and variables are declared just as in a program or procedure block. Procedures and functions are presented as headings without bodies.

```
export
  const
    pi = 3.14159;
  type
    polar = record
               radius,theta: real
            end;
  var
    scalefactor: real;
    origin: polar;
  function makepolar (a: complex): polar;
  procedure setorigin (a: complex);
```

The export part may make use of things in turn exported from other modules listed in the import part (such as the type "complex"). Every module must have an export part.

- The **implement** part consists of the reserved word IMPLEMENT, followed by constant, type, variable, procedure and function declarations, followed by the word END. All the procedures and functions whose headings were in the export part must be present in their entirety in the implement part. The implement part may make use of things in turn exported from other modules listed in the import part.

  A module does not have to export procedures or functions, it may be used simply to create data types or variables. In such a case there will be nothing between the words IMPLEMENT and END.

A complete module, "complexmath", is shown on the next page. It has no import part because it depends on no other modules. (The module is also on the DOC: disc; the source is called CXMODULE.TEXT).

The import and export parts are said to define the module's **interface** to other modules or programs. This interface is public: the information it contains is available to any importer of the module.

The implement part is said to be "private", which means that everything between the words IMPLEMENT and END is hidden from importers. Anything declared here is unknown outside the module, except for procedures and functions whose headings were also included in the export part.

The private and public parts of the module are separated in this way so that its implement part can safely be changed without altering programs or other modules which import it. This independence of modules from programs is a key to developing software libraries. Another implication is that modules can only be dependent on other modules, not on programs. The reason is simply that there's no way to import a program into a module (since programs have no export declarations).

It was stated at the outset that a module is a "fragment" of a program. To be more precise, a module is a set of global (outer level) declarations which can be compiled once, then bound into a program by an IMPORT declaration in that program.

```
 1:D        0 module complexmath;
 2:D        1 export
 3:D        1   type
 4:D        1     complex = record
 5:D        1                       re: real;
 6:D        1                       im: real;
 7:D        1                 end;
 8:D        1   const
 9:D        1     zero = complex [re:0.0,im:0.0];
10:S
11:D        1   function equal (a,b: complex): boolean;
12:D        1   function add   (a,b: complex): complex;
13:D        1   function mul   (a,b: complex): complex;
14:D        1   function dvd   (a,b: complex): complex;
15:D        1   function conj  (a: complex):   complex;
16:D        1   function mag   (a: complex):   real;
17:D        1   function scmul (scale:real; a:complex): complex;
18:S
19:D        1 implement
20:D        1
21:D  -32   1   function equal (a,b: complex): boolean;
22:C        2     begin   equal := (a.re=b.re) and (a.im=b.im)   end;
23:S
24:D  -32   1   function add (a,b: complex): complex;
25:C        2     begin   add.re := a.re+b.re; add.im := a.im+b.im   end;
26:S
27:D  -32   1   function mul (a,b: complex): complex;
28:C        2     begin
29:C        2       mul.re := (a.re*b.re-a.im*b.im);
30:C        2       mul.im := (a.re*b.im+a.im*b.re);
31:C        2     end;
32:S
33:D  -32   1   function dvd (a,b: complex): complex;
34:D  -40   2   var  denom: real;
35:C        2   begin
36:C        2     denom := sqr(b.re)+sqr(b.im);
37:C        2     if denom = 0.0 then halt(-5); (*divide by zero*)
38:C        2     dvd.re := (b.re*a.re + b.re*a.re) / denom;
39:C        2     dvd.im := (b.re*a.im - b.im*a.re) / denom;
40:C        2   end;
41:S
42:D  -16   1   function conj (a: complex): complex;
43:C        2     begin   conj.re :=  a.re; conj.im := -a.im   end;
44:S
45:D  -16   1   function mag (a:complex): real;
46:C        2     begin   mag := sqrt(sqr(a.re)+sqr(a.im))   end;
47:S
48:D  -24   1   function scmul (scale:real; a:complex): complex;
49:C        2     begin
50:C        2       scmul.re := scale*a.re;
51:C        2       scmul.im := scale*a.im
52:C        2     end;
53:S
54:C        1 end. (*complexmath*)
```

No errors.

Developing and Testing a Module

The Workstation environment supports a structured approach to the development and testing of software modules. This is important because modules often become part of the system library, and many programs may depend on them. The usual steps in the development cycle are:

- Decide what the module will do — define its functionality. Write the interface part first, specifying what other modules will be needed and what things the module will export. Remember that when the finished module is imported into a program, only this interface will be "visible". Figure out how a program will use the exported things to get the module to do its job.

- Decide how the module will be tested. Write a test program which will thoroughly exercise it.

- Write the implement part of the module. Embed the completed module in the test program, and compile the two together. Leave the module inside the program until you're satisfied with the results.

- Extract the module from the test program. This can be done by using the Librarian to pull it out of the compiled test program, or by separating the module's source text with the Editor and compiling it independently.

- Use the compiled module. It can be put in your LIBRARY, or left as a user library which is manually linked to dependent programs, or loaded into memory by the Permanent load command.

The following listing shows the source of CXMODULE embedded into the program called CX (also on the DOC: disc):

```
Pascal [Rev 3.2  1/15/87] CX0.TEXT              19-Oct-82 09:15:51 Page 1

      1:D        0 program cx (listing);
      2:S
      3:D        1 module complexmath;
      4:D        1 export
      5:D        1    type
      6:D        1       complex = record
      7:D        1                      re: real;
      8:D        1                      im: real;
      9:D        1                  end;
     10:D        1    const
     11:D        1      zero = complex [re:0.0,im:0.0];
     12:S
     13:D        1    function equal (a,b: complex): boolean;
     14:D        1    function add   (a,b: complex): complex;
     15:D        1    function mul   (a,b: complex): complex;
     16:D        1    function dvd   (a,b: complex): complex;
     17:D        1    function conj  (a: complex):   complex;
     18:D        1    function mag   (a: complex):   real;
     19:D        1    function scmul (scale:real; a:complex): complex;
     20:S
```

```
21:D        1 implement
22:D        1
23:D   -32  1    function equal (a,b: complex): boolean;
24:C        2        begin  equal := (a.re=b.re) and (a.im=b.im)   end;
25:S
26:D   -32  1    function add (a,b: complex): complex;
27:C        2        begin  add.re := a.re+b.re; add.im := a.im+b.im   end;
28:S
29:D   -32  1    function mul (a,b: complex): complex;
30:C        2        begin
31:C        2          mul.re := (a.re*b.re-a.im*b.im);
32:C        2          mul.im := (a.re*b.im+a.im*b.re);
33:C        2        end;
34:S
35:D   -32  1    function dvd (a,b: complex): complex;
36:D   -40  2    var   denom: real;
37:C        2    begin
38:C        2      denom := sqr(b.re)+sqr(b.im);
39:C        2      if denom = 0.0 then halt(-5); (*divide by zero*)
40:C        2      dvd.re := (b.re*a.re + b.re*a.re) / denom;
41:C        2      dvd.im := (b.re*a.im - b.im*a.re) / denom;
42:C        2    end;
43:S
44:D   -16  1    function conj (a: complex): complex;
45:C        2        begin  conj.re :=  a.re; conj.im := -a.im   end;
46:S
47:D   -16  1    function mag (a:complex): real;
48:C        2        begin  mag := sqrt(sqr(a.re)+sqr(a.im))   end;
49:S
50:D   -24  1    function scmul (scale:real; a:complex): complex;
51:C        2        begin
52:C        2          scmul.re := scale*a.re;
53:C        2          scmul.im := scale*a.im
54:C        2        end;
55:S
56:C        1 end; (*complexmath*)
57:S
58:S
59:S
60:S
```

```
61:D          1 import complexmath;
62:S
63:D          1 const
64:D          1   pi = 3.141592654;
65:D          1   nsteps = 16;
66:D          1 var
67:D    -32   1   a,b: complex;
68:D   -304   1   table: array [1..nsteps+1] of complex;
69:D   -320   1   theta,thetastep: real;
70:D   -324   1   i: integer;
71:D   -324   1   listing : text;
72:S
73:C          1 begin
74:C          1   theta := 0.0;
75:C          1   thetastep := pi/(2*nsteps);
76:C          1   a := zero;   b := zero;
77:C          1   for i := 1 to nsteps+1 do
78:C          2     begin
79:C          2       a.re := sin(theta);   (*leave im part zero*)
80:C          2       b.im := cos(theta);   (*leave re part zero*)
81:C          2       table[i] := add(a,b);
82:C          2       theta := theta + thetastep;
83:C          2     end;
84:C          1   writeln(listing,'     REAL     ',
85:C          1                   '  IMAGINARY ',
86:C          1                   '  MAGNITUDE ');
87:C          1   for i := 1 to nsteps+1 do
88:C          2     writeln(listing,' ',
89:C          2             table[i].re,' ',table[i].im,' ',
90:C          2             mag(table[i]) );
91:C          1 end.
```

No errors.

```
     REAL          IMAGINARY       MAGNITUDE
0.00000E+000   1.00000E+000    1.00000E+000
9.80171E-002   9.95185E-001    1.00000E+000
1.95090E-001   9.80785E-001    1.00000E+000
2.90285E-001   9.56940E-001    1.00000E+000
3.82683E-001   9.23880E-001    1.00000E+000
4.71397E-001   8.81921E-000    1.00000E+000
5.55570E-001   8.31470E-001    1.00000E+000
6.34393E-001   7.73010E-001    1.00000E+000
7.07107E-001   7.07107E-001    1.00000E+000
7.73010E-001   6.34393E-001    1.00000E+000
8.31470E-001   5.55570E-001    1.00000E+000
8.81921E-000   4.71397E-001    1.00000E+000
9.23880E-001   3.82683E-001    1.00000E+000
9.56940E-001   2.90285E-001    1.00000E+000
9.80785E-001   1.95090E-001    1.00000E+000
9.95185E-001   9.80171E-002    1.00000E+000
1.00000E+000  -2.0510E-010     1.00000E+000
```

An Illustration

The accompanying listing shows the module "complexmath" embedded in a test program. The test program isn't very thorough, since it only checks the constant "zero" and the "add" and "mag" functions.

Modules embedded in a program may be intermixed with global constant, type, and variable declarations, but all modules must appear before any of the program's global procedures and functions. Usually all the modules are put first, followed by the program's own globals. If there are several modules, they must be ordered so that no module is imported by another (or by the program) until it has been declared.

Notice the semicolon following the END of the module (line 56), and that the program must have an IMPORT declaration (line 61) even though the module is physically present in the program.

Program "cx" can be compiled and run as shown. If you'd like to try it, invoke the Compiler by pressing ⌐C⌐ at the Main Command Level. When asked what text to compile, put the diskette labelled DOC: in a drive and answer:

   DOC:CX **Return** or ⌐ENTER⌐

Let the Compiler put the output file on the same disc (accept the default output file).

Compiling a Module Separately

The file generated by compiling CX.TEXT is a library with two modules: the main program "cx" and module "complexmath". Strictly speaking a program isn't a module, but within a library it has a directory entry just as if it were. You might wish to use the Librarian and see for yourself. The Librarian can display every detail of a code file. Had there been several modules in "cx", each one would have had a separate directory entry.

It's important to be clear about the distinction between modules and libraries. A library is a file, created by the Compiler, Assembler, or Librarian. The library's name is its file name, which you can see with the Filer. Inside the library is a directory naming all the modules in that file. The library directory can only be displayed by the Librarian.

If you were satisfied at this point with the testing of "complexmath," you could use the Librarian to pull that one module out of the code file and either make it a user library or add it to the system library. The Librarian documentation describes how to do this.

Another alternative is to compile the module separately. Simply use the Editor to create a text file having only the module. Notice that when the module is compiled alone, it must be followed by a period instead of a semicolon. The Compiler will also accept a sequence of several modules, separated by semicolons. The last one must be followed by a period. The program listing on the next page shows the listing generated by a separate compilation.

How the Compiler Finds Library Modules

A module which has been compiled is called a "library module." Library modules can be imported by programs or other modules, because the compiled code file carries with it a description of the module's interface. The Compiler is able to read this description and, from it, determine how to properly access everything exported by the module.

When the Compiler processes an IMPORT declaration, it must find the modules named in the import list and read their interface specifications. A particular search pattern is followed, which is repeated for each module named in the list.

- If the imported module has been previously declared or imported in the source text being compiled, then the reference is to that module.

- If no module of that name has been found, the Compiler must search library files on mass storage. The files to be searched may be specified by a $SEARCH$ option. See the subsequent Compiler Options section of this chapter.

- If there is no $SEARCH$ option or the module is not found in the specified list of files, the Compiler goes on to look in the system library.

- If the module still isn't found, error 104 (undeclared identifier) is issued.

Note

The Compiler **does not** search libraries which have been loaded into memory with the P-load command. Module interface specifications are not retained with memory-resident libraries.

A module which is imported may itself import other modules, which are listed in its import section. The Compiler must follow such a chain all the way back to its root, to a module which imports no others. The search pattern just described is applied recursively, to a maximum depth of ten levels. For a restriction, see the subsequent "INCLUDE Files" section. Sometimes in following an import chain, a module is named in more than one import list. The Compiler actually reads the interface specification for a module just once.

If a program imports module "A", which in turn imports module "B", the things exported from "B" are nevertheless hidden from the program. To make them visible, "B" must also be imported into the program.

The listing below shows program "cx" recompiled to search for module "complexmath" in a library called "CXMODULE" on mass storage unit #3. The second listing shows "cx" recompiled assuming "complexmath" has been put into the current System Library.

```
 1:D        0 program cx (listing);
 2:S
 3:D        1 $search '#3:CXMODULE'$
 4:D        1 import complexmath;
 5:S
 6:D        1 const
 7:D        1   pi = 3.141592654;
 8:D        1   nsteps = 16;
 9:D        1 var
10:D   -32  1   a,b: complex;
11:D  -304  1   table: array [1..nsteps+1] of complex;
12:D  -320  1   theta,thetastep: real;
13:D  -324  1   i: integer;
14:D  -324  1   listing : text;
15:S
16:C        1 begin
17:C        1   theta := 0.0;
18:C        1   thetastep := pi/(2*nsteps);
19:C        1   a := zero;  b := zero;
20:C        1   for i := 1 to nsteps+1 do
21:C        2     begin
22:C        2       a.re := sin(theta);  (*leave im part zero*)
23:C        2       b.im := cos(theta);  (*leave re part zero*)
24:C        2       table[i] := add(a,b);
25:C        2       theta := theta + thetastep;
26:C        2     end;
27:C        1   writeln(listing,'     REAL     ',
28:C        1                   '   IMAGINARY ',
29:C        1                   '   MAGNITUDE ');
30:C        1   for i := 1 to nsteps+1 do
31:C        2     writeln(listing,' ',
32:C        2             table[i].re,' ',table[i].im,' ',
33:C        2             mag(table[i]) );
34:C        1 end.
```

No errors.

```
 1:D        0 program cx (listing);
 2:S
 3:D        1 import complexmath;   (* from LIBRARY *)
 4:S
 5:D        1 const
 6:D        1   pi = 3.141592654;
 7:D        1   nsteps = 16;
 8:D        1 var
 9:D   -32  1   a,b: complex;
10:D  -304  1   table: array [1..nsteps+1] of complex;
11:D  -320  1   theta,thetastep: real;
12:D  -324  1   i: integer;
13:D  -324  1   listing : text;
14:S
15:C        1 begin
16:C        1   theta := 0.0;
17:C        1   thetastep := pi/(2*nsteps);
18:C        1   a := zero;  b := zero;
19:C        1   for i := 1 to nsteps+1 do
20:C        2     begin
21:C        2       a.re := sin(theta);  (*leave im part zero*)
22:C        2       b.im := cos(theta);  (*leave re part zero*)
23:C        2       table[i] := add(a,b);
24:C        2       theta := theta + thetastep;
25:C        2     end;
26:C        1   writeln(listing,'    REAL    ',
27:C        1                   '  IMAGINARY ',
28:C        1                   '  MAGNITUDE ');
29:C        1   for i := 1 to nsteps+1 do
30:C        2     writeln(listing,' ',
31:C        2             table[i].re,' ',table[i].im,' ',
32:C        2             mag(table[i]) );
33:C        1 end.
```

No errors. No warnings.

How the Loader Finds Library Modules

When the Compiler processes an import declaration, it does not copy, or in any other way bind, the library module into the program being compiled. Instead it emits reference information (called REF's) which enable the loader or linker to make the required connections later. Usually REF's are satisfied (hooked up to the library module) at the last possible moment: when you Run the program.

A compiled program contains no record of where the Compiler found any imported modules. The loader has a search pattern it uses to find imported things the program needs:

- First, the file being loaded is searched. There may be modules in it which were compiled at the same time as the program.

- Then memory-resident libraries are searched. The memory-resident libraries are those you have loaded with the P command, the contents of INITLIB (which is automatically loaded at boot time), and the modules of the Operating System itself. The order of search is most-recently-loaded first.

- Finally, the current System Library is searched. If a required module is in the System Library, then it will be loaded with the program and will remain in memory until a different program is executed.

- If there are still unresolved references, the loader reports them on the CRT. The program won't run. Control is returned to the Main Command Level.

If your program only imports from the System Library, everything is taken care of automatically. This is the most common case. If the program imports from user libraries via the $SEARCH$ option, then you must help out the loader in one of three ways:

- Use the P-load command to load copies of the libraries into memory before running the program. Do this just once, because the P-load command does not check to see if modules have already been loaded! Memory-resident libraries stay there until you re-boot.

- Use the Librarian to make a new library containing the compiled program and any modules it needs. This new library is an unlinked, executable program. It will be linked automatically when it is loaded.

- Use the Librarian to link the necessary modules to the program. The resulting library is a linked, executable program. It will probably still have some unresolved references (for instance to the system read and write routines), which will be resolved at load time.

A Subtle Point

The loader doesn't search for modules, it searches for **external names**. Each procedure or function exported has an external name, as do most structured constants. A single name is used for all the variables a module exports; it is actually the name of a place in memory where storage for the variables will be allocated. Certain things, such as types and simple constants, are only useful at compile time and so have no external name.

If two differently named modules each define the same load-time name, there is no problem because external names created by the Compiler identify the module where the name originated. However, if both modules have the same name and define the same load-time name, the most recently loaded copy **overrides** the older one.

Since some module names are used by the Operating system, you should avoid using these names for your own modules unless you **intend** to overide the name of a system entry point. These names are listed in the Technical Reference Appendix.

Although two modules that contain the same symbol can be loaded, they cannot both be imported by the same program without conflicts.

$INCLUDE Files

The source text of a module or program can be broken up into several text files which are edited separately but compiled as a group. The $INCLUDE Compiler option tells the Compiler to insert the text of another file into the one it is presently compiling.

```
program showinclude (input,output);
$include 'MYVOL:DECLARS'$
$include 'SYSVOL:BODY'$
end.
```

If the required volume is not online when needed, the Compiler pauses and prompts you to insert the proper volume.

Miscellaneous:

- An included file may in turn include another file. This "nesting" is allowed to a maximum depth of 10.

- Importing a library module is a form of file inclusion, and counts against the maximum allowable depth of 10 while the import declaration is being processed.

- If the imported module has an import declaration in its own interface, the Compiler will follow the chain and find those module interfaces too. This is another form of nested file inclusion.

Note

IMPORT "attaches" only code files whereas INCLUDE "attaches" or inserts only source files.

What Can Go Wrong?

This section discusses some problems which may occur when using the Compiler, and how to solve them.

Can't Run the Compiler

1. If the system reports, `Cannot open 'CMP:COMPILER'`, the volume with the Compiler is not on-line. You may have removed the volume and not put it back. If the Compiler wasn't found when the system booted, you are expected to put the CMP: disc (which contains the Compiler) on-line.

2. If the system reports, `Cannot load 'COMPILER'`, either the disc medium is bad or not enough memory is installed in the Computer to run the Compiler. It is desirable to have at least 393 Kbytes; the system is normally sold with at least 524 Kbytes.

Errors 900 thru 908

During compilation, three files are written by the Compiler: the code file, which is the one you want, and the REF and DEF files. The latter two are temporary working storage for linkage information which is appended to the code file if the compilation terminates normally. All three of these files are normally opened on the same volume (the volume to which you directed the code file).

Each of these files is subject to three classes of error:

- Error in opening the file.

- Insufficient space to open the file.

- File fills up before compilation finishes.

An error in opening the file usually means the volume is not on-line. It can also indicate that the volume's directory is full.

The amount of space allocated to the code file on a LIF or WS1.0 file system is usually half of the largest free area on the volume, with the potential to expand to the second half of that area if needed. If you get errors 900, 903, or 906, then you need to make more room on the volume to which the code file was directed, or use a different volume.

The REF file by default is opened with 30 blocks of disc space on the same volume as the code file. A Compiler option at the beginning of the source program can change the size and the volume selected for REF. There's no simple rule which gives the "right" size for the REF file. If the file fills up (error 907), then make it bigger in proportion to the amount of program that remained to compile when the error occurred.

`$REF 50$`	Allocate 50 blocks
`$REF 'CHARLIE:'$`	Put it on volume CHARLIE
`$REF 'CHARLIE:', REF 50$`	Put it on CHARLIE and allocate 50 blocks

Errors When Importing Library Modules

1. Syntax errors in the interface of an imported library module. This usually indicates that the library module itself tried to import some other module which was not found by the Compiler's search algorithm.

2. Errors 608, 610: Include or import nesting too deep. If module "A" imports "B", which imports "C" and so forth, the Compiler must follow the chain to its end. The chain can only be 10 imports deep. Since the same file handling mechanism is also used to process $INCLUDE files, the combined limit on import and inclusion nesting is 10 deep.

3. Error 613: Imported module does not have interface text. If the library has been linked by the Librarian, the interface specification has been removed. Also, a main program looks internally like a module; but it has no interface text.

Not Enough Memory

If the Compiler generates error −2 (not enough memory), then there isn't enough room in memory to compile the program. You can watch the numbers which appear on the screen in square brackets as the compilation proceeds — they show approximately how much memory is left. There are two primary reasons for running out of memory during a compilation. One of them is large procedure bodies, and the other is P-loaded files.

Large Procedure Bodies

When the Compiler processes a procedure, the entire procedure (declarations and body) is scanned. An internal representation of the procedure, called a "tree", is built. This tree is not complete until the scanner reaches the end of the procedure, and only then does code generation begin. The tree form takes a lot of storage, particularly the statements making up the body. If you write a procedure whose body is ten pages long, the Compiler is very likely to run out of memory. The moral is that you should keep your procedures reasonably short. A good guideline is that no procedure should be longer than a page or two.

P-loaded Files

If you've Permanent-loaded many libraries or programs, or space has been allocated to a memory-resident mass storage volume, you may reboot the system to recover the memory, and try again.

Insufficient Space for Global Variables

You may discover, either at compile time or at run time, that there isn't sufficient space for the global variables of your program. If this happens, please refer to "Implementation Dependencies" in the *Pascal Language Reference* manual, which explains the limitations and what to do if you exceed them.

Errors 403 thru 409

These errors should never be reported. They indicate a malfunction in the Compiler itself. If this ever happens, please show the program which causes it to your HP field support contact.

Error 154: Illegal argument to match pass-by-reference parameter.

The HP Pascal Language Standard specifies that elements of packed structures (packed arrays or records) cannot be passed as arguments to `var` or `anyvar` parameters. This rule was not enforced in versions of the Pascal compiler prior to revision 3.1 (3.1 enforced it for `var` parameters by default). Revision 3.2 added enforcement of the rule for `anyvar` parameters. If sources that compiled previously now produce error 154 for arguments to `var` or `anyvar` parameters, use the `$ALLOW_PACKED ON$` compiler directive to relax this restriction. The object code generated for the call will be the same as that generated by previous revisions of the compiler.

Compiler Options

Compiler options affect the code that is emitted by the Compiler. For instance, the $DEBUG ON$ option causes the Compiler to emit a MC68000 TRAP instruction after the object code for each Pascal statement, allowing you to single-step a program.

Sometimes there are restrictions on where an option may appear.

Location	Restriction
Anywhere:	No restriction. Indicates that the location of the option in the file is irrelevant.
At front:	Applies to entire source file; must appear before the first "token" in the source file (before PROGRAM, or before MODULE if compiling a list of modules).
Not in body:	Applies to a whole procedure or function; can't appear between BEGIN and END. Good practice to put these options immediately before the word BEGIN, or the procedure heading.
Statement:	Can be applied on a statement-by-statement basis or to a group of statements, by enabling before and disabling after the statements of interest.
Special:	As explained under the particular option.

If an option appears in the interface (import or export) part of a module, it will have effect as the module is compiled. However, the option itself will not become part of the interface specification (export text) in the compiled module's object code.

ALIAS

Default: External name = Procedure Name

Location: Special; see "Location" below

This option specifies a name, other than the name used in the Pascal procedure or function declaration, to be used by the loader.
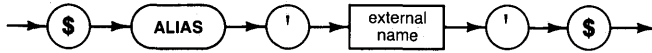
Item	Description	Range
external name	string	1 to 80 ASCII characters

Semantics

The string parameter specifies the external name for the procedure in whose header the option appears.

Location

The option must appear between the keywords PROCEDURE or FUNCTION and the first symbol following the semicolon (;) denoting the end of the procedure or function declaration.

The option may **not** appear in an export section.

Example

```
procedure $alias 'charlie'$ p (i: integer);  external;
```

Within the program, calls use the name "p"; but the loader will link to the external name "charlie" wherever "p" is found.

ALLOW_PACKED

Default: OFF

Location: Anywhere

This option permits or prohibits the passing of elements of packed arrays or records to var and anyvar[1] parameters. It also permits or prohibits using the sizeof function on elements of packed arrays or records[2].
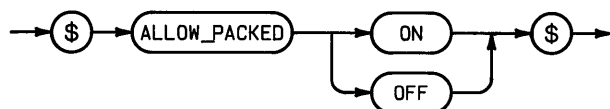
Semantics

"ALLOW_PACKED" is interpreted as "ALLOW_PACKED ON".

Passing elements of packed arrays or records to var or anyvar parameters is illegal in HP Standard Pascal, but the Workstation Pascal Compilers prior to Version 3.2 allowed it. Pascal 3.1 and subsequent compilers allow passing of packed elements to var parameters *only* if the compiler option ALLOW_PACKED is ON.

ON specifies that elements of packed structures will be allowed to be passed to var and anyvar parameters in functions and procedures. You may need to add the option $ALLOW_PACKED ON$ and re-compile existing pre-3.1 Pascal source code to run it on the 3.1 system. Pascal 3.1 sources may require recompilation to run on 3.2 if they use "system internals".

OFF specifies that passing elements of packed structures to var or anyvar parameters is illegal. Attempts to do so result in a compile-time error message 154: "Illegal argument to match pass-by-reference parameter".

OFF also specifies that passing elements of packed structures to sizeof is illegal. Attempts to do so result in a compile-time error message 125: "Erroneous type of argument for built-in routine".

Note

Pre-3.1 compilers allowed only certain packed elements to be passed to var parameters. These are the elements which ALLOW_PACKED affects. Others, which pre-3.1 compilers forbade from being passed, are still forbidden in 3.1 and later compilers.

---

[1] The restriction on passing elements of packed structures to anyvar parameters is new for Pascal Workstation 3.2. See the description of the SYSPROG directive in this chapter for information on anyvar. Further references to anyvar are in the "Workstation Implementation" section of the *HP Pascal Language Reference*.

[2] The restriction on passing elements of packed structures to sizeof parameters is new for Pascal Workstation 3.2. See the description of the SYSPROG directive in this chapter for information on sizeof. Further references to sizeof are in the "Workstation Implementation" section of the *HP Pascal Language Reference*.

ANSI

Default: OFF

Location: At Front

This Compiler option selects whether an error message is to be emitted for use of any feature of HP Standard Pascal not contained in ANSI/ISO Standard Pascal.
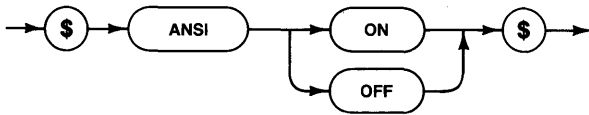
Semantics

"ANSI" is interpreted as "ANSI ON".

ON causes error messages to be issued for use of any feature of HP Standard Pascal which is not part of ANSI/ISO Standard Pascal. If the error is issued, no code file will be emitted.

OFF suppresses the error messages.

Example

```
$ansi on$
```

CALLABS

Default: ON

Location: Anywhere

This Compiler option determines whether 16-bit relative or 32-bit absolute jumps are to be generated by the compiler.

```
→($)→(CALLABS)──┬──(ON)──┬→($)→
                └──(OFF)──┘
```
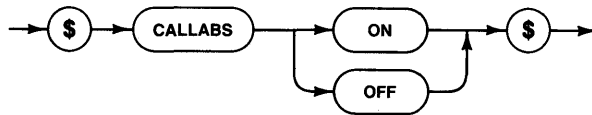
Semantics

"CALLABS" is interpreted as "CALLABS ON".

ON specifies that 32-bit absolute jumps will be emitted for all forward and external procedure calls.

OFF specifies 16-bit PC-relative jumps.

This option is allowed on a statement-by-statement basis.

Example

```
$callabs off$
```

CODE

Default: ON

Location: Not in Body

This Compiler option is used to control whether a CODE file will be generated by the Compiler.
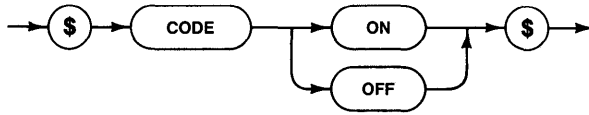
Semantics

"CODE" is interpreted as "CODE ON".

ON specifies that executable code will be emitted and placed in a CODE file; OFF specifies that no code will be emitted and no file is to be generated.

Example

```
$code off$
```

CODE_OFFSETS

Default: OFF

Location: Not in Body

This Compiler option controls the inclusion of program counter offsets in the compiler listing.
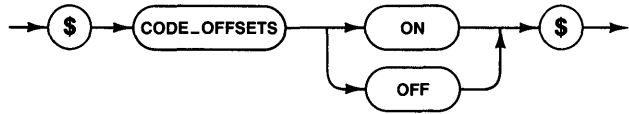
Semantics

"CODE_OFFSETS" is interpreted as "CODE_OFFSETS ON".

ON specifies that line-number/program-counter pairs will be printed for each executable statement listed. This can be applied on a procedure-by-procedure basis.

Example

```
$code_offsets on$
```

COPYRIGHT

Default: Not Applicable

Location: Anywhere

This Compiler option is provided for inclusion of copyright information.

Item	Description	Range
copyright message	string	Entire copyright must fit on one line.

Semantics

The string parameter is placed in the object file as the owner of the copyright. If more than one COPYRIGHT option is included, the last one is effective.

Example

```
$copyright 'Hewlett Packard Company, 1983'$
```

DEBUG

Default: OFF

Location: Not in Body

This Compiler option controls whether the code produced by the Compiler contains the additional information necessary for full use of the debugger.

```
→($)→(DEBUG)→┬→(ON)→┬→($)→
             └→(OFF)→┘
```
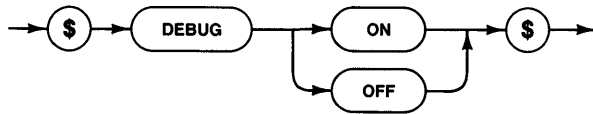
Semantics

"DEBUG" is interpreted as "DEBUG ON"

"DEBUG ON" will cause debugging instructions to be emitted for the procedure bodies following it. It may be applied on a procedure−by−procedure basis.

Example

```
procedure buggy;
var  i: integer;
$debug on$
begin
   . . .
end;
$debug off$
```

DEF

Default: 10 512-byte blocks (on same volume as code output)

Location: At Front

This Compiler option allows you to change the size and location of the temporary Compiler file named ".DEF".
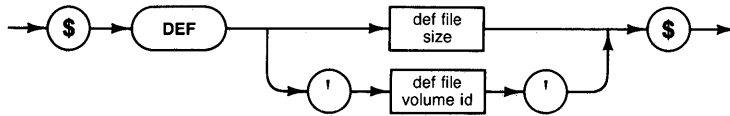
Item	Description	Range
def file size	integer constant	less than 32 767
def file volume specification	literal	valid volume specification

Semantics

If the parameter is a string, it specifies the volume where a temporary Compiler file called ".DEF", which holds external definitions, will be stored. If the parameter is a number, it specifies how many 512-byte blocks will be allocated for the DEF file. See the preceding explanation of "What Can Go Wrong?" for further information.

Examples

```
$def 50$
$def 'DEFVOL:'$
$def 'ANYVOL:', def 50$
```

FLOAT_HDW

Default: OFF (in COMPILER) ON (in COMPILE20)

Location: Not in body

This Compiler option enables and disables the use of floating-point hardware.
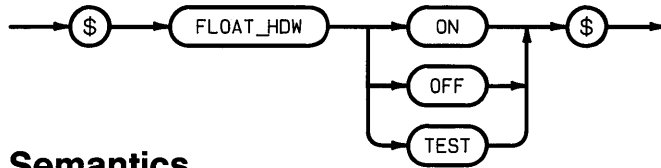
Semantics

HP 98635 Floating-point Math Card

The HP 98635 is an optional PC board that increases the execution speed of floating-point math computations. This board can be installed in all Series 200 computers.

- ON instructs COMPILER to generate accesses to 98635 hardware for the floating-point operations listed below. If the hardware is not installed when the program is executed, an error will be reported.

- OFF tells COMPILER to generate math library calls for floating-point operations.

- TEST causes COMPILER to generate both hardware accesses and library calls. The code includes tests for the presence of floating-point hardware. If the test succeeds (at execution time), the hardware accesses are used; otherwise, the library calls are used.

Operations that can potentially use the 98635 floating-point card include: addition, subtraction, multiplication, division, negation, and the sqr function. Hardware can also be used by any operation that converts an integer into a real or longreal; however, hardware is not used by operations that convert reals or longreals into integers. All other math functions call library routines. The math library will use the 98635, where appropriate, if it is present.

MC68881 or MC68882 Floating-point Math Co-processor

When using this option with the COMPILE20 compiler, it has a slightly different meaning.

- ON causes COMPILE20 to generate MC68881 or MC68882 co-processor instructions. The object code generated can only run on Series 300 computers equipped with the optional MC68881 or MC68882 math co-processor.

- OFF causes COMPILE20 to generate code that uses Pascal math libraries.

- TEST is not allowed (COMPILE20 reports an error).

Operations that can can potentially use the MC68881 or MC68882 hardware include all floating-point math computations except trunc. The math library will not attempt to use the MC68881 or MC68882 even if one present.

Note

If you are writing interrupt service routines (ISRs), see the *Pascal 3.2 Procedure Library*, "System Devices" chapter, "Interrupt Processing Overview" section.

HEAP_DISPOSE

Default: OFF

Location: At Front

This Compiler option enables and disables "garbage collection" in the heap.
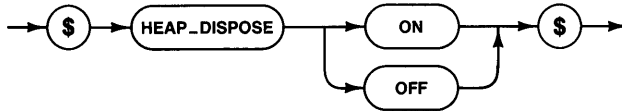
Semantics

"HEAP_DISPOSE" is interpreted as "HEAP_DISPOSE ON"

ON indicates that DISPOSE allows disposed objects to be reused.

OFF does not recycle disposed objects.

If enabled, this option must appear at the front of the **main program**.

Example

```
$heap_dispose on$
program recycle;
  ...
begin
  new(p);
  dispose(p);   (*free up cell*)
  new(p);       (*probably gets same cell back*)
end.
```

The HEAP_DISPOSE option must be the same (either ON or OFF) in the program and in **all** modules imported by the program. Erroneous results may occur if those declarations don't agree, because there is no way for the Compiler to check on which option other modules have used.

IF

Default: Not Applicable

Location: Anywhere

This Compiler option allows conditional compilation.
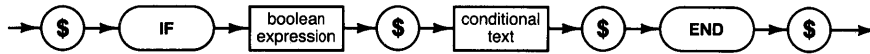
Item	Description	Range
boolean expression	expression that evaluates to TRUE or FALSE	may only contain compile-time constants
conditional text	Pascal source text to be conditionally compiled	—

Semantics

If the expression evaluates to FALSE, then text following the option is skipped up to the next END option.

If the boolean evaluates to TRUE, then the text following the option is compiled normally.

IF-END option blocks may not be nested.

String constants may not be used.

Example

```
const  fancy = true;
       limit = 10;
       size = 9;
...
$if fancy and ((size+1)<limit)$
   ...    (* this will be skipped *)
$end$

$if FALSE$
   ...    (* this will also be skipped. *)
$end$
```

INCLUDE

Default: Not Applicable

Location: Anywhere

This Compiler option allows text from another file to be included in the compilation process.
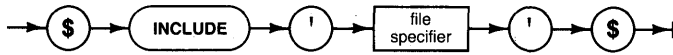
Item	Description	Range
file specifier	string	any valid file specifier

Semantics

The string parameter names a file which contains Pascal source to be included at the current position in the program. Included code may contain additional INCLUDE options (nesting level is 10). The remainder of the line containing this option must be blank except for the closing "$".

Example

```
program inclusive;
  $include 'source:declare'$
  $include 'source:body'$
end.
```

IOCHECK

Default: ON

Location: Statement

This Compiler option enables and disables error checking following calls to file system I/O routines.
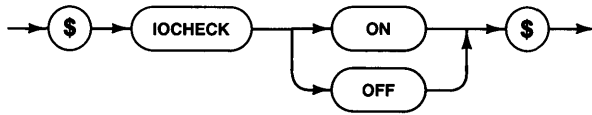
Semantics

"IOCHECK" is interpreted as "IOCHECK ON"

ON specifies that error checks will be emitted following calls on file system I/O routines such as RESET, REWRITE, READ, WRITE. This option can be used in conjunction with the standard function IORESULT if the UCSD or SYSPROG language extensions have been enabled. IOCHECK can be specified on a statement-by-statement basis.

OFF specifies that no error will be reported in case of failure.

Example

```
$ucsd$
...
$iocheck off$
reset(f,'datafile');
$iocheck on$
if ioresult <> 0 then writeln('IO error');
```

LINENUM

Default: Not Applicable

Location: Anywhere

This Compiler option allows the user to establish an arbitrary line number value.
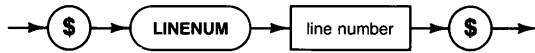
Item	Description	Range
line number	integer constant	1 thru 65 535

Semantics

The integer parameter becomes the current line number (for listing purposes, and debugging purposes).

Example

    $linenum 20000$

LINES

Default: 60 lines per page

Location: Anywhere

This Compiler option allows the user to specify the number of lines-per-page on the compiler listing. 2000000 lines-per-page suppresses autopagination.
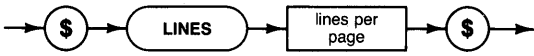
Item	Description	Range
lines per page	integer constant	20 thru MAXINT

Examples

```
$lines 55$
$lines 2000000$    (*suppress auto-pagination*)
```

LIST

Default: ON to PRINTER:

Location: Anywhere

This Compiler option controls whether or not a listing will be generated, and to where it will be directed.
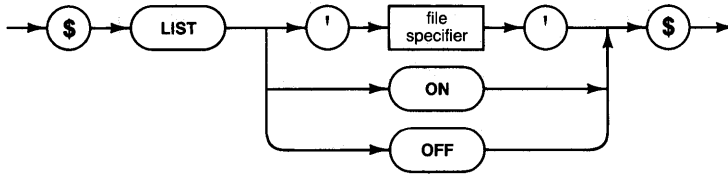
Item	Description	Range
file specifier	string	any valid file specifier

Semantics

"LIST" is interpreted as "LIST ON".

LIST with a file specifier specifies that the file is to receive the compilation listing.

LIST OFF suppresses listing.

LIST ON resumes listing. No listing will be produced at all, regardless of this option, unless requested by the operator when the Compiler is invoked.

Examples:

```
$list 'MYVOL:KEEPLIST'$
$list 'PRINTER:'$
$list off$
```

OVFLCHECK

Default: ON

Location: Statement-by-statement

This Compiler option gives the user some control over overflow checks on arithmetic operations.
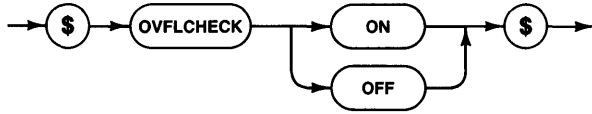
Semantics

"OVFLCHECK" is interpreted as "OVFLCHECK ON"

ON specifies that overflow checks will be emitted for all in-line arithmetic operations.

OFF does not suppress all checks; they will still be made for 32-bit integer DIV, MOD, and multiplication.

Example

```
$ovflcheck off$
```

PAGE

Default: Not Applicable

Location: Anywhere

This Compiler option causes a form-feed to be sent to the listing file if compilation listing is enabled.

→($)→(PAGE)→($)→

Example

```
$page$
```

PAGEWIDTH

Default: 120

Location: Anywhere

This Compiler option allows the user to specify the width of the compilation listing.
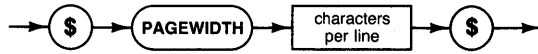
Item	Description	Range
characters per line	integer constant	80 thru 132

Semantics

The integer parameter specifies the number of characters in a printer line.

Example

```
$pagewidth 80$
```

PARTIAL_EVAL

Default: OFF

Location: Statement-by-statement

This Compiler option enables or disables the partial evaluation of multiple logical operations.
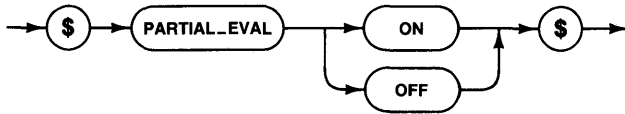
Semantics

"PARTIAL_EVAL" is interpreted as "PARTIAL_EVAL ON".

ON suppresses the evaluation of the right operand of the AND operator when the left operand is FALSE. The right operand will not be evaluated for OR if the left operand is TRUE.

OFF causes all operands in logical operations to be evaluated regardless of the condition of any other operands.

Example:

```
$partial_eval on$
while (p<>nil) and (p^.count>0) do
  p := p^.link;
```

RANGE

Default: ON

Location: Statement-by-statement

This Compiler option enables and disables run-time checks for range errors.

```
──($)──(RANGE)──┬──(ON)──┬──($)──►
                └──(OFF)──┘
```
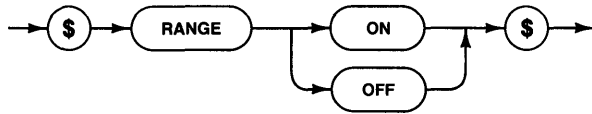
Semantics

"RANGE" is interpreted as "RANGE ON".

ON specifies that run-time checks will be emitted for array and case indexing, subrange assignment, and pointer dereferencing.

Example

```
var a: array[1..10] of integer; i: integer;
...
i := 11;
$range off$
a[i] := 0;     (* invalid index not caught! *)
```

REF

Default: 30 512-byte blocks (on same volume as code output)

Location: At Front

This Compiler option allows you to change the size and location of the temporary compiler file named ".REF".
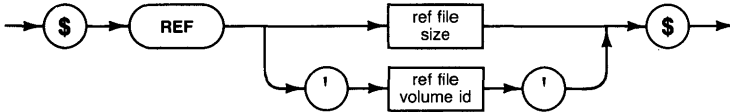
Item	Description	Range
ref file size	integer constant	less than 32767
ref file volume specification	string literal	any valid volume specification

Semantics

If the parameter is a string, then it specifies the volume where a temporary Compiler file named ".REF", which holds external references, will be stored. If the parameter is a number, it specifies how many 512-byte blocks will be allocated for the REF file. See "What Can Go Wrong, Errors 900 to 908".

Examples

```
$ref 20$
$ref 'JUNKVOL:'$
$ref 'JUNKVOL:', ref 50$
```

SAVE_CONST

Default: ON

Location: Anywhere

This Compiler option controls whether the name of a structured constant may be used by other structured constants.
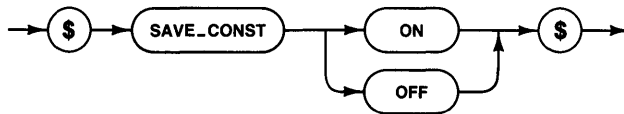
Semantics

"SAVE_CONST" is interpreted as "SAVE_CONST ON".

ON specifies that compile-time storage for the value of each structured constant will be retained for the scope of the constant's name (so that other structured constants may use the name).

OFF specifies that storage will be deallocated after code is generated for the structured constant.

Example

```
$save_const off$
type ary = array [1..100] of integer;
const acon = ary [345,45691, ..... ];
   (*big constants take lots of compile-time memory*)
```

SEARCH

Default: Not Applicable

Location: Anywhere

This Compiler option is used to specify files to be used to satisfy IMPORT declarations.
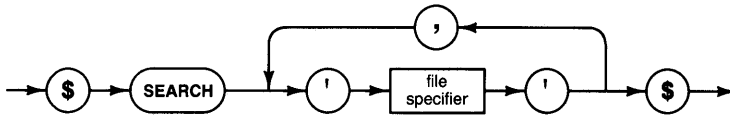
Item	Description	Range
file specifier	string	any valid file specifier

Semantics

Each string specifies a file which may be used to satisfy IMPORT declarations. Files will be searched in the order given. The current system library file is always searched last. A maximum of 9 files may be listed.

Multiple SEARCH options are allowed; for instance, you may want to use one for each import declaration. Note that only the last one encountered during compilation will be in effect for any import statement (i.e., these options are not cumulative).

Example

```
$search 'FIRSTFILE','SECONDFILE'$
```

SEARCH_SIZE

Default: 10 files

Location: At front

This Compiler option allows you to increase the number of external files you may SEARCH during a module's compilation.
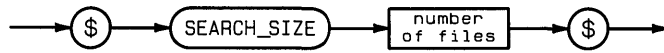
Item	Description	Range
number of files	integer constant	less than 32 767

Semantics

When compiling a Pascal module, it is sometimes desirable to import another module from another file. To import a module from another file, the SEARCH option is used to identify the file. Up to 10 SEARCH options may be given unless the SEARCH_SIZE option is given. The SEARCH_SIZE option allows you to SEARCH up to 32 766 external files for imported modules.

Example

```
$search_size 30$
```

STACKCHECK

Default: ON

Location: Not in Body

This Compiler option enables and disables stack overflow checks.
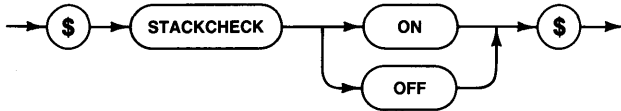
Semantics

"STACKCHECK" is interpreted as "STACKCHECK ON".

ON specifies that stack overflow checks will be generated at procedure entry. It is very dangerous to turn overflow checks off! Obscure and unreported errors may result.

Example

```
$stackcheck off$
procedure unsafe;
var
  may_smash_heap: array [1..500] of integer;
begin ... end;
```

SWITCH_STRPOS

Default: OFF

Location: Anywhere

This Compiler option reverses the positions of the parameters of the STRPOS function.
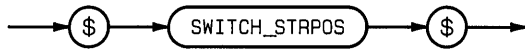
Semantics

When this Compiler option is used, the expected order of the parameters is that of the HP standard. In Series 200/300 Pascal (like UCSD's POS function), the STRPOS function expects the first string parameter to be the search pattern and the second string parameter to be the source string in which the search takes place. Later the HP standard was established with the order of the parameters reversed. If $WARN OFF$ is not in effect, then the Compiler issues a harmless warning that you are not conforming to the standard. If you wish to conform to the standard, give the SWITCH_STRPOS option in your program.

Example

    $switch_strpos$

SYSPROG

Default: System Programming Extensions not enabled

Location: At Front

This Compiler option makes available some language extensions which are useful in systems programming applications.

Semantics

$SYSPROG$ is interpreted as $SYSPROG ON$.

See the *Pascal Language Reference*, "Workstation Implementation" section for information on Language extensions.

Example

```
$sysprog$
PROGRAM machine_dependent;
 ...
```

TABLES

Default: OFF

Location: Not in Body

This Compiler option allows you to turn the listing of symbol tables on or off.
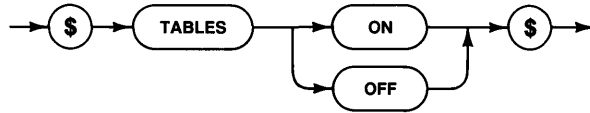
Semantics

$TABLES$ is interpreted as $TABLES ON$.

ON specifies that symbol table information will be printed following the listing of each procedure. This is useful for very low-level debugging.

Example

```
$tables$
procedure hasabug (var p: integer);
var
   . . .
```

UCSD

Default: UCSD not enabled

Location: At Front

This Compiler option allows the compiler to accept most UCSD Pascal language extensions.

```
→($)→(UCSD)→($)→
```

Semantics

See the *Pascal Language Reference*, "Workstation Implementation" section for information on Language extensions.

Example

```
$ucsd$
program funnyio;
var
  f: file;    (* no type specified! *)
begin
  unitread(8,ary,80,10);
end.
```

WARN

Default: ON

Location: At Front

This Compiler option allows the user to suppress the generation of compiler warning messages.
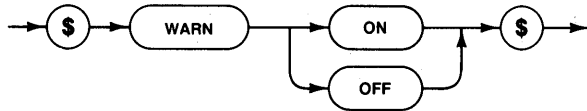
Semantics

$WARN$ is interpreted as "WARN ON" and compiler warnings will be issued.

Example

```
$warn off$
```

How Pascal Programs Use the Stack

This section describes how Pascal programs use the stack to store data, return addresses for procedures, and pointers needed to access variables belonging to nested procedures. The information can be useful when writing Assembler language routines, and when debugging at the machine level.

You can also investigate this subject by writing some Pascal test programs and then looking at the emitted code with the Librarian's Unassemble command. Two Compiler options also produce valuable information: $DEBUG ON$ correlates the machine code displayed by Unassemble with the original Pascal lines, and $TABLES$ causes the Compiler to print a description of the size and location of each object in the program.

The Pascal Stack

Seven types of data can be stored on the stack:

- procedure/function parameters
- return addresses
- local variables
- stack frame pointers
- static links
- addresses used by the with statement
- function results

Two address registers are reserved for stack manipulations:

- A7 - the stack pointer (SP)
- A6 - the stack frame pointer (SF)

The stack grows downward in memory (toward smaller addresses) as procedures are called, with A7 always pointing to the base (beginning, lowest address) of the datum on the "top" of the stack. That is, when space is allocated for a procedure which has been called, the area allocated has a lower (more negative) address than the space already allocated for the calling procedure. Space allocated to a procedure is called its **stack frame**.

However, variables extend upward in memory. This simply means that the address of the first element of an array, or the first field of a record, is lower than the address of the second element or field.

Global Variables

Register A5 is reserved as the global base register. A reference to any program or module global variable is always formulated as a displacement from where register A5 points. The maximum size of the global area is 65,536 bytes (the displacement field size). In practice, not all of this space is available to the application. Some of this area is used for system globals, command interpreter globals, permanently loaded programs and modules, and so forth. Also there is a limit of 32 766 bytes of globals for any program or module.

See the Assembler chapter for details on how to reference Pascal global variables from Assembler-language programs.

Procedure Calls

Here is a brief summary of how one procedure calls another. The calling procedure first pushes parameters to be passed to the called procedure onto the stack. The calling procedure then executes a JSR instruction, which pushes the return address on the stack and jumps to the entry point of the called procedure.

Each parameter is pushed on the stack by first decrementing the stack pointer (A7) an amount equal to the size of the parameter, then storing the parameter where SP now points. (Pushing a byte decrements the stack pointer by two, since SP must always have an even value.)

The first instruction executed by the called procedure is a LINK instruction (if $STACKCHECK OFF$). The LINK instruction format and function is illustrated below:

```
format :   LINK A6,#d

function:  A6 -> -(SP)  --  push the stack frame pointer
                            onto the stack
           SP -> A6     --  set the stack frame pointer equal
                            to the stack pointer
           SP-d -> SP   --  drop the stack by the size(d) of the
                            local variables for the called procedure
```

If the program is compiled with $STACKCHECK ON$ (which is the default), a TRAP #1 instruction is issued instead of LINK. The trap service routine checks for stack overflow as it adjusts A6 and SP. In this case the size #-d is stored in the next word after the TRAP #1 instruction.

The stack frame pointer (A6) is used by the called procedure to reference its local variables and passed parameters. See Figure 6 for an illustration of stack usage for procedure calls textually nested at level 1. Level 1 procedures are those declared at the global level of a program or module.

If the called procedure is not textually nested at level 1, the calling procedure pushes a pointer to the stack frame of the procedure in which the **called** procedure is declared. This pointer is called the **static link**. It is used by the called procedure to resolve references to intermediate variables — variables which are neither local to the called procedure, nor globals of the program.

An example might help to clarify the static link. Consider the following program structure (indentation indicates nesting):

```
program main
    procedure p1
        procedure p2
            procedure p3
            procedure p4
```

Assume this calling sequence: main calls p1, p1 calls p2, p2 calls p4. If p4 calls p3 then the static link pushed would be that of procedure p2 (since p4 is declared within p2). If instead p4 were to call p2, then the static link would point to p1 (p2 is nested within p1). See Figure 7 for a detailed example of static links.

Returning Control to the Calling Procedure

The called procedure is responsible for stack cleanup and for effecting the return to the calling procedure. Any parameters, local data, or static links belonging to the called procedure must be removed from the stack before returning to the caller. Once this is complete, a return to the calling procedure can be performed.

The stack cleanup is performed in two steps:

1. Restore the stack frame pointer. Use the UNLK instruction to remove local data from the stack.

   ```
   Format:    UNLK A6

   Function:  A6 -> SP     --  Set the stack equal to the stack
                               frame pointer.
              (SP)+ -> A6 --  Load the stack frame pointer from
                               the stack and autoincrement the stack
                               pointer (this leaves the stack
                               pointer pointing to the return
                               address).
   ```

2. Restore the stack pointer. This removes the static link and parameters from the stack. After this step, the stack pointer should be as it was before the procedure call.

   The called procedure returns to the caller by branching to the return address. If the return address was saved in an address register during stack cleanup then an indirect JMP through the address register is executed. If the return address was left on the stack then an RTS instruction is executed.

   ```
   Format :   RTS

   Function:  (SP)+ -> PC   --  Set the program counter to the
                                value pointed to by the stack pointer
                                and pop the value off the stack.
   ```

See Figure 8 for an example of a return from a called procedure.

Function Calls

Function calls differ from procedure calls only in that they return results. The result is usually returned on the stack. It is the responsibility of the calling procedure or function to pop the result off of the stack. This is normally done when the result is referenced.

Parameter Passing Mechanisms

There are two kinds of formal parameter: those passed by reference, and those passed by value.

reference parameters	all handled alike
value parameters	a) simple value parameters simple types (integer, char.) array and record types <= 4bytes
	b) copied value parameters: reals, and array and record types > 4 bytes

Reference parameters are those which are specified VAR in the procedure heading. They are "passed by reference": the address of the actual parameter is passed to the called procedure or function. This address is used for all references to the parameter. No copying of the parameter is performed.

Value parameters are those which are **not** specified VAR in the heading. They are "passed by value": a copy of the parameter is passed to the called procedure or function. If the value parameter is a simple type (except REAL) then its value is pushed on the stack. If the parameter is a simple REAL, or an array or record (and its size is more than 4 bytes), then its address is pushed on the stack by the caller. Before the called routine executes its first statement, it uses the pushed address to copy the parameter into its local data space (the Compiler reserved this space in addition to the local variable space).

Values of type "procedure" are not copied; their values are pushed directly even though they are eight bytes long.

Function Results

Sometimes the calling environment must allocate temporary space in which to return function results. In general this is necessary when the function returns a result which is bigger than 4 bytes. The temporary space is allocated as part of the program's global area if the call is from the main program; otherwise it is allocated as part of the local data area. The amount of temporary space required is determined at compile time. Functions which return a value of type real are an exception; the result area is on the stack and occupies eight bytes.

Return Address	
A6	
Local Var_1	local variables for calling procedure
Local Var_n	
Parm_1	parameters for called procedure
Parm_n	
Return Address	return address pushed on the stack by the jsr instruction executed by the calling procedure
A6 → A6	the stack frame pointer pushed on the stack by the link instruction executed by the called procedure
Local Var_1	local variables for called procedure
A7 → Local Var_n	

Figure 6. Pascal Procedure Calls (Without Static Links)

(The stack is pictured growing toward the bottom of the page. Pointers actually address the "bottom" of the designated entry.)

The following Pascal program illustrates the use of the static link.

```
$DEBUG ON$
program main(input,output);

var i:integer;

    procedure A;
    var k:integer;

        procedure B;
        var m:integer;

            procedure C(i:integer);
            var o:integer;

                procedure D;
                var q:integer;
                begin
                    i := k;
                    k := m;
                    m := o;
                    o := q;
                    q := 1;
                    B;
                    C(i);
                end; {D}

            begin {C}
                m := o;
                D;
            end; {C}

        begin {B}
            k := 1;
            C(m);
        end; {B}

    begin {A}
        B;
    end; {A}

begin {main}
    A;
end.
```

Note

The preceding program is only for the purpose of illustrating the use of the static link. Running the program results in error -2 (not enough memory), because the program recurses indefinitely.

Consider the following calling sequence:

> Main calls A
>> A calls B
>> B calls C (with m as the parameter)
>> C calls D
>> D calls B

The stack for this calling sequence is shown in Figure 7.
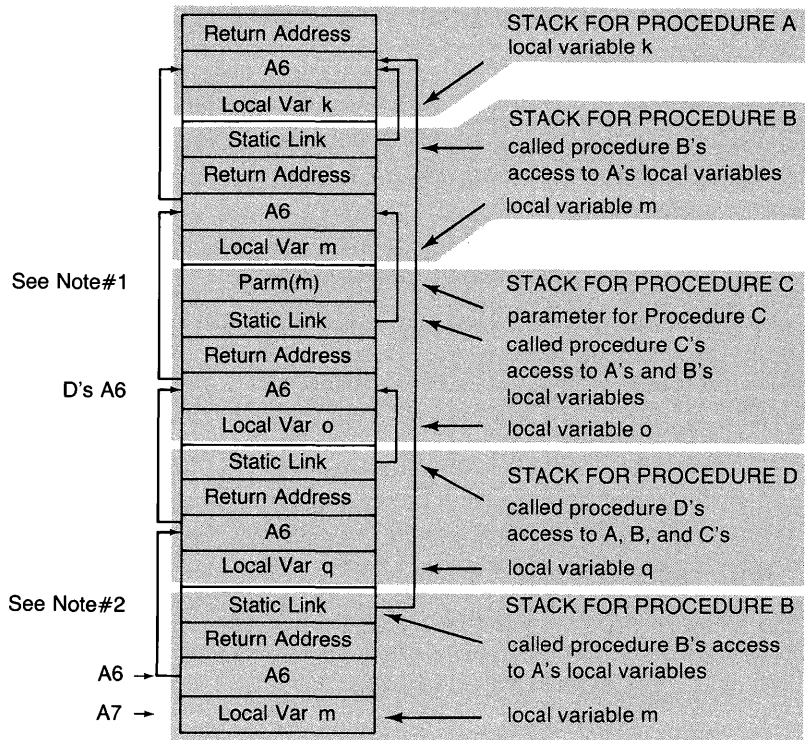
Figure 7. Pascal Procedure Calls (With Static Links)

(Pointers actually address the "bottom" of the designated entry.)

Note 1: The static link and the parameters are always accessed at positive offsets from A6. The effective address of the static link (if present) is always 8(A6). Local variables are at negative displacements from A6.

Note 2: In general, the static link gives the called procedure access to the intermediate variables of procedures which precede it in the calling sequence. In this particular case, the static link gives procedure B access to variables declared within procedure A.

Procedure D reaches intermediate variables using:

its current stack frame pointer

the difference between its nesting level and that of the called procedure

In this case procedure D is at level 4 and procedure B is at level 2, for a relative distance of 2. Therefore procedure B must follow two static links to reach the stack frame of B.

In other words:

```
MOVEA.L  8(A6),A0        — Get procedure D's static link.
MOVEA.L  8(A0),A0        — Get procedure C's static link.
MOVE.L   8(A0),-(SP)     — Get procedure B's static link
                           and push it on the stack.
```

Remember: procedure C's static link gives access to B's locals, procedure B's static link gives access to A's, etc.

Note 3: When nested procedures reference intermediate variables, they use the static link. An example of this is when procedure D references k and m in the statement k := m;.

k is declared in procedure A and m is in procedure B. The nesting level relative to procedure D for k is 3 and for m it is 2.

The following code will perform the statement: k := m.

```
MOVEA.L  8(A6),A0        — Get D's static link.
MOVEA.L  8(A0),A0        — Get C's static link.
MOVEA.L  8(A0),A0        — Get B's static link.

MOVEA.L  8(A6),A1        — Get D's static link.
MOVEA.L  8(A1),A1        — Get C's static link.

MOVE.L   -4(A1),-4(A0)   — Store m in k.
```

The return to procedure A (as shown in the following stack segment) is accomplished in four steps. Note: the register prefixes indicate the value of the register for the indicated step. The values are those the registers have AFTER the step has been executed.

```
Step 1: UNLK A6           — Restore the stack frame pointer (A6)
Step 2: MOVEA.L (SP)+,A0  — Save the return address in A0
Step 3: ADDQ.L #12,SP     — Restore the stack pointer
Step 4: JMP (A0)          — Return to where procedure A called B
```

If there are not any parameters then the return sequence normally is:

```
UNLK A6                   — Restore the stack frame pointer.
MOVEA.L (SP)+,(SP)        — Replace the static link with the return address.
RTS                       — Return to procedure A.
```

If there is no static link (and no parameters) then the sequence is:

```
UNLK A6 — Restore the stack frame pointer.
RTS     — Return to procedure A.
```
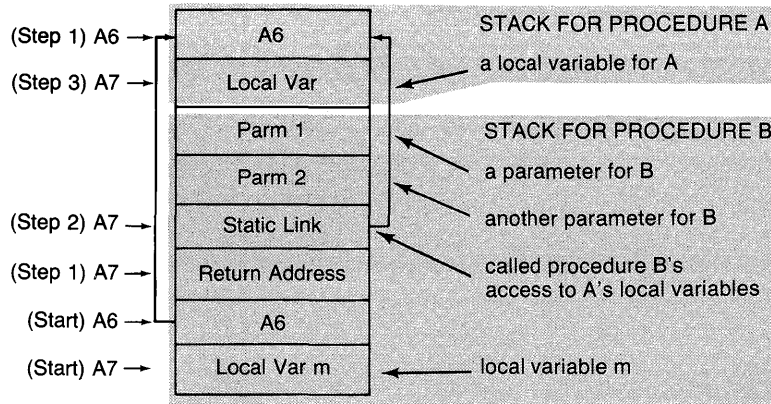
Figure 8. Return From a Procedure Call

(Pointers actually address the "bottom" of the designated entry.)

WITH Addresses

When a procedure includes a with statement that dereferences a pointer (e.g. with x^ do) the compiler allocates space at the bottom of the procedure's stack frame, below local variables, for temporary storage of the pointer's value. The size is 4 bytes per simultaneously active with using dereferencing.

The Assembler

7

Introduction

This chapter describes the use of the Workstation Assembler subsystem. The Assembler translates assembly language routines into object code which can be executed on this system. Assembly language programming gives you the ability to optimize critical sections of a program (primarily for reductions in execution time or code size).

The Workstation Assembler, which has a two-pass design, translates source files written in the assembly language specified by Motorola in the MC68000[1], MC68020, and MC68030 manuals. It also assembles instructions for the MC68881 and MC68882 floating-point math co-processor, which are described in the MC68881 manual. A summary of the syntax required by this Assembler is provided in the "Instruction Syntax" section of this chapter.

Although it is not a tutorial on assembly language programming, this chapter contains the information necessary to write and execute assembly language routines on the Workstation System. The first section demonstrates the method of generating external procedures and entire object modules of Assembler code that can be interfaced to Pascal programs. You should be familiar with the concept of Pascal modules before attempting to emulate them in assembly language; refer to the "Pascal Compiler" chapter for pertinent information.

Unlike most other assembler subsystems you may have used, the directives ("pseudo-operations") that you give to the Workstation Assembler are specified within the source program — they are not given in an interactive session with the Assembler. The Workstation Assembler's pseudo-operations are fully described in the "Pseudo-op Reference" section near the end of this chapter.

---

[1] The MC68000 manual also includes documentation for the MC68010 processor.

Operating the Assembler

This section shows you how to:

- Invoke the Assembler

- Specify the name of your text file program and your resulting code file

- Give listing specifications

- Interpret the listing

Invoking the Assembler

The Assembler is delivered on the ASM: disc. If you plan to run the Assembler several times in a session, you can use the Permanent command to keep the Assembler in memory ready to run. Otherwise, put the ASM: disc in a disc drive and press ⬚A⬚ to run the Assembler.

Source File Specification

If there is a work file (see the Filer chapter), that file will be automatically assembled and there will be an "errors only" listing on the CRT. If the "errors only" listing is sufficient your source program file can be specified as the work file . Otherwise, clear the work file.

If there is no work file, you will be prompted to enter the name of your program file:

```
What source file?
```

Enter the volume name (unless using the default volume, explained in the Filer chapter) and file name of your source program. It is not necessary to include the ".TEXT" suffix of your file name. If it is not included, it will be added for you by the Assembler subsystem. For example, if your program file is named PROGRAM.TEXT and it is on the volume named TOMS, then use this file specification:

```
TOMS:PROGRAM
```

Listing File Information

You are then prompted to specify whether or not you will want a listing of the assembly:

```
Do you want a program listing (y/n/e) ?
```

You may type:

⬚Y⬚ for a complete listing
⬚N⬚ for no listing but errors reported on the CRT
⬚E⬚ for a listing of the errors only

If you want a listing, you can have it printed immediately or have the Assembler generate a file of the listing information:

```
What listing file (default PRINTER:PROGRAM.ASC) ?
```

For a printer listing, press `Return` or `Enter`.

To generate a listing on a file, enter the name of the volume and of the file. It is recommended that a size specification be made for the listing file (see the Filer chapter) when using LIF or WS1.0 file systems. Otherwise, the largest space on the disc will be reserved for the listing, which may leave no space for the code file. A good rule of thumb is to use twice the number of blocks used by program file. For example, if TOMS:PROGRAM.TEXT is 20 blocks long, a size specification of 40 blocks is made for the listing file.

```
TOMS:PROGLIST.TEXT[40]
```

It is possible to have a CRT screen listing by specifying "CONSOLE:" as the listing file. This is not recommended unless the program is very small, or an "error only" listing was requested. The listing will be scrolled onto the screen and you are returned to the Main Command Level. There is no way to control the screen listing.

Object File Specification

Finally, you are prompted to give a name for the code file that will be generated by the Assembler. The default name is that of the source file with the suffix ".CODE" replacing ".TEXT".

```
Output file (default is TOMS:PROGRAM.CODE) ?_
```

If the default name is acceptable, press `Return` or `Enter`. If you want to specify another name, enter the complete file specification.

After this entry, the Assembler begins processing your program. The CRT displays when the first pass of the Assembler is completed, along with the number of errors encountered during the first pass. There is a similar display for second pass. After the second pass is completed, you are returned to the Main Command Level. If no errors were generated during the assembly, then a code file was created.

If the assembly program is executable (has a start address), you run it by pressing `R` at the Main Level. The Run command will run your program automatically until:

- another program is assembled or compiled.
- a workfile is specified.
- the computer is powered down.
- the system volume is re-specified.

If the Run command no longer works for your program, then use the eXecute command and specify the name of the code file that was generated.

Interpreting the Listing

The output from the Assembler contains the following information. The first column on the listing indicates the (decimal) number of the source-program line. For each line of input, a line number is generated. This is true of blank lines as well.

The second column shows the current location counter (relative to the code origin). The value is in hex notation unless the DECIMAL pseudo-op is specified. When the program is loaded, the number in column two can be added to the base address of the load to obtain the absolute address of the instruction. This is useful information when debugging.

The third column shows the hexadecimal (base 16) representation of the machine code for the instruction or value of equated symbols generated by the Assembler.

The right side of the listing is a copy of the source program.

Sample Assembler Output

Line Number	Address	Emitted Code	Source Code
36	00000000		rorg 0
37			
38	00000000	0000 0000	simple2_zero dc.l 0,0
38	00000004	0000 0000	
39			
40	00000008	4E41	simple2_initialize trap #1 (stack check)
41	0000000A	0000	dc.w 0 (no local space)
42			
43	0000000C	4CFA 0300 FFF0	movem.l simple2_zero,a0-a1
44	00000012	48ED 0300 FFF0	movem.l a0-a1,sum(a5)
45			
46	00000018	4E5E	unlk a6
47	0000001A	4E75	rts
48			
49	0000001C	4E41	simple2_partadd trap #1
50	0000001E	FFFC	dc.w -4
51			
52		0000 0010	result equ 16
53		0000 000C	x equ 12
54		0000 0008	y equ 8 (relative to a6)
55		0000 0004	ret_addr equ 4

Error messages are listed under the line in which they occur. At the completion of the assembly, the total 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.

The Programming System

It is assumed that you will be writing most of your programs in Pascal. In the instance where the execution speed of a particular routine is insufficient, this section will show you how to translate the Pascal routine into an Assembler language routine and call it from your Pascal program.

It is possible to write a simple procedure, put it in the "system library" (usually a file named LIBRARY on the * volume), and access it with an EXTERNAL directive from the Pascal program. However, add some interface text to the routine, and you have created a module. The benefits of modules are that global variables and constants may be used for communication among modules. Special types which define parameters need only be declared in the module containing the called procedure.

A Pascal module was developed for use as an example. The Librarian was used to disassemble the code into its Assembler language counterpart. The intent of this section is to explain the method of interpreting the disassembly information and producing a working Assembler language module. The examples are also available on the documentation disc (DOC:). The file (ASMB_P1) imports the file (file ASMB_M1). These are both Pascal files. The Pascal file (ASMB_P2) imports the Assembler language file (ASMB_M2).

You'll notice in the example program that the variables are declared to be of the type which are defined in the imported module. If the program merely declared one or two of the procedures to be EXTERNAL procedures, those special types would have to be defined in every program that called the procedures. It would be like going to the Library for a book and having to write down the table of contents every time you wanted to use the book.

For your Assembler language module to interface cleanly with the Pascal program, the conventions of the Compiler must be followed. That is, you must set up the Assembler language module to act as if it were a compiled Pascal module. You must also exit the module leaving everything in order, as a Pascal module would.

The information you need to accomplish a clean "Pascal-to-Assembler language" interface is presented in this section. You should understand how the Compiler:

- Prepares interface text (IMPORT text)
- Declares entry points (DEF table)
- Declares external references (EXT table)
- Passes parameters
- Creates global variable space
- Initializes modules
- Recovers from errors
- Returns from subroutines

You will find a listing of the Pascal program and module as originally written, a listing of the disassembly of the module, and a listing of the final, working Assembler language module. These listings are included at the end of this section. Remove them from the manual and keep them out for reference as you're reading this material.

The first subject covered is the method of generating the IMPORT text. This is what separates an importable module from a simple EXTERNAL routine. The subsequent material is of concern in either case. There will be a short explanation of the method for declaring EXTERNAL routines toward the end of the section.

The IMPORT Text

Certain information must be passed from an imported module to the Compiler to complete the module interface. This information is the IMPORT (or "interface") text. Actually, IMPORT text contains IMPORT declarations and EXPORT declarations. It's called IMPORT text here because it's what the Compiler needs when it is importing the module. It must know the module name, global variables, global constants, and procedure and function names. If special TYPE declarations are needed to define the variables, they must be included in this information.

At compile time, your imported Assembler module must make this information available to the Compiler. This is done with the SRC pseudo-op. See how the IMPORT text of the Pascal listing is exactly the same as the SRC-IMPORT text below.

```
src module simple2;
src export
src    type
src       rec = record
src             i1 : integer;
src             i2 : integer;
src          end;
src    const
src       zero = rec[i1:0,i2:0];
src    var
src       lastresult : rec;
src    procedure initialize;
src    procedure add(a,b : rec;
src                   var out : rec);
src end;
```

The SRC section does not actually name the module or get the global space. There are separate techniques for accomplishing these things, which are discussed later.

The DEF Table

The DEF table contains the locations of all the entry points in the Pascal module and the location of its global space. This information is provided for the linking loader. The information is used to link all the modules together before they can be loaded and executed.

```
DEF table of 'SIMPLE';
    SIMPLE                      Gbase
    SIMPLE_ADD                  Rbase+82
    SIMPLE_INITIALIZE           Rbase+10
    SIMPLE_SIMPLE               Rbase+252
    SIMPLE_ZERO                 Rbase
```

The symbol "SIMPLE" which is the same as the module name, is the name of the module's global variable space. This symbol is entered into the DEF table automatically when you reserve the global space using the COM statement. This is explained later in the global variable section of this chapter.

"SIMPLE_ADD" AND "SIMPLE_INITIALIZE" are the entry points into the two procedures "add" and "initialize". When writing assembly language routines, they must be named as the Compiler names its procedures. The Compiler appends the module name to the front of the procedure name, separated by an "_". When the Compiler looks at your IMPORT section, it assumes that the procedures have been named by its convention. When it's time for the loader to hook everything together, it looks for those procedure names in your module's DEF table.

"SIMPLE_SIMPLE" is the entry point, or location, of the module initialization body. Module initialization is discussed later in this chapter.

"SIMPLE_ZERO" is the location of the structured constant, "zero", which appears in the IMPORT section of the module. Any code which resides in the assembly module and is declared in the IMPORT section of the module, must appear in the DEF table. It, too, must be named by prefixing the module name to the constant name that you declare in the IMPORT section. This name must appear as a label at the constant's location in the program.

You must create a DEF table for the assembly version of your routine. This is done using the DEF statement. Notice that all the symbols in the Pascal module's DEF table are named in the DEF statements below except the symbol for the global variable space. The global variable symbol is entered into the table at the time the space is reserved with the COM statement.

```
def simple2_add
def simple2_initialize
def simple2_zero,simple2_simple2
```

The EXT Table

The EXT table that you get from the Librarian is the list of the symbols that the loader must find in some corresponding DEF table so our module can access those external items.

```
EXT table of 'SIMPLE';

    SYSGLOBALS
```

"SYSGLOBALS" is the only symbol in this particular list. We need to access some of the system's global variables in our routine so we must know where they are kept. They are in the global variable space for the system, "Sysglobals". (See the TRY-RECOVER section for more details about the system globals)

The EXT table is created in the assembly module using REFA and REFR. Both instructions enter symbol names into the EXT table. REFA causes the symbol to be referenced using absolute addressing. REFR causes the symbol to be referenced using 16-bit PC relative addressing. See REFA, REFR, SMODE and LMODE in the pseudo-op reference section.

In the example, "Sysglobals" was declared as external using REFA.

If other modules' global variable sections were to be referenced, the symbol for those areas would also need to be included in our EXT table. This is explained in the global variable section.

Declaring the Module Name

The module is named using MNAME. This puts the name of the module in the module directory for the Compiler to reference when importing the module.

If no MNAME is used, the module name will be the same as the file name.

Passing Parameters

When parameters are passed to a procedure, the values or addresses of variables in the parameter list are pushed onto the stack. The function result space is put on the stack if the routine is a function. The leftmost variable in the parameter list is pushed onto the stack. Then the rest are pushed onto the stack in order from left to right. The return address is pushed onto the stack automatically by the processor at the time the JSR instruction is encountered.

For example:

```
114    2F0E              move.l a6,-(sp)
116    487A  005A        pea  Rbase+208
120    2B4F  FFF6        move.l sp,SYSGLOBALS-10(a5)
124    598F              subq.l #4,sp
126    2F2E  FFF0        move.l -16(a6),-(sp)
130    2F2E  FFF8        move.l -8(a6),-(sp)
134    4EBA  FF98        jsr  Rbase+32
138    2B5F  FFF8        move.l (sp)+,Gbase-8(a5)
142    598F              subq.l #4,sp
144    2F2E  FFF4        move.l -12(a6),-(sp)
148    2F2E  FFFC        move.l -4(a6),-(sp)
152    4EBA  FF86        jsr  Rbase+32
156    2B5F  FFFC        move.l (sp)+,Gbase-4(a5)
160    202D  FFF8        move.l Gbase-8(a5),d0
164    D1AD  FFF0        add.l d0,Gbase-16(a5)
```

The stack is mapped in the following way:

```
              ┌──────────────────┐
              │ FUNCTION RESULT  │
12(SP)→       ├──────────────────┤
              │   VALUE of x     │
 8(SP)→       ├──────────────────┤
              │   VALUE of y     │
 4(SP)→       ├──────────────────┤
              │ RETURN ADDRESS   │
  (SP)→       └──────────────────┘
```
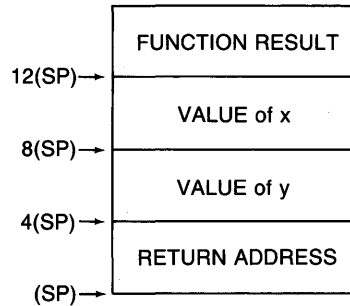
Notice that the stack grows downward (toward smaller addresses).

If a parameter is passed by reference, a 4-byte address is pushed onto the stack. When passing by value, values up to 4-bytes are pushed onto the stack, but larger values are essentially passed by reference. That is, a 4-byte address is pushed on the stack. In this case, a copy of the value must be made in local variable space so that the actual parameter is not altered. This is illustrated in the Local Variable section.

More information can be found in the Compiler chapter under the heading "How Pascal Programs Use the Stack".

Declaring Global Variables

You must understand how the Compiler allocates global variable space so that you get and use global space the same way. The value stored in register A5 is the base address for all global areas. Each module that declares global variables is allocated an area for them. The symbol assigned to the area is equated to the distance from the base address in A5 to the area. Globals are then referenced symbolically, relative to A5.

The name for the location of a module's globals (relative to the address in A5) is the same as the module name. So the symbol for the global area for "module simple" would be "simple".

Determine how much space you need for your globals. When determining how much space is needed, you must also consider any variables that are internally global to the module. Notice on the Pascal module listing that the variable, "sum" is global to the module.

If you are rewriting a Pascal module as we have done in the example, the Compiler provides variable size information beside the variable declarations on the listing (the negative number). More detailed information can be displayed using the Compiler's $TABLES$ directive. You must specify the amount using a negative value also. Declare global space using the COM statement:

```
COM simple2,-16
```

The value, −16, corresponds to the global variables, "lastresult" and "sum". Both are records containing two integers each.

The COM statement also enters the symbol into the DEF table.

Referencing Global Variables

The Assembler module name is SIMPLE2 as is its global base. Notice in the DEF Table that "SIMPLE" is equal to "Gbase" (Global BASE) for the Pascal module. Global locations in the disassembly are referenced using the symbol "Gbase" rather than "simple".

```
DEF table of 'SIMPLE':

    SIMPLE                      Gbase
    SIMPLE_ADD                  Rbase+82
    SIMPLE_INITIALIZE           Rbase+10
    SIMPLE_SIMPLE               Rbase+252
    SIMPLE_ZERO                 Rbase
                  .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    170   202D FFFC     move.l Gbase-4(a5),d0
    174   D1AD FFF4     add.l d0,Gbase-12(a5)
    178   4E76          trapv
    180   206E 0008     movea.l 8(a6),a0
    184   4CAD 1E00     movem.w Gbase-8(a5),a1-a4
```

When writing your assembly language module, use the COM symbol to reference globals. The Assembler doesn't recognize "Gbase". In our assembly module, the global variables are referenced using "SIMPLE2".

```
lastresult       equ simple2-8
lastresult_i1    equ simple2-8
lastresult_i2    equ simple2-4
sum              equ simple2-16   (all are relative to a5)
sum_i1           equ simple2-16
sum_i2           equ simple2-12
escapecode       equ sysglobals-2
recovery_rec     equ sysglobals-10
```

Note

When structured variables are used, the individual elements of the structure are referenced at progressively higher addresses within the structure's space.

If, for example, you had declared two integers separately instead of together in one record, you would refer to them as:

```
lastresult_i1          EQU simple-4
lastresult_i2          EQU simple-8
```

Referencing Other Module's Globals

When referencing the global variables of another module, it is necessary to establish the external reference using REFR or REFA.

The individual variables are referenced at negative offsets from the symbol and relative to A5, as described in the global variable section above. As was mentioned previously, offsets into data areas are provided on Compiler listings.

Local Variables

There are several methods for getting local variable space. The following method is recommended for those intending to produce purely relocatable code. This is important if the code is to be committed to ROM.

Notice that the first instruction in each of the disassembled routines is:

```
TRAP #1
```

TRAP #1 calls a system routine which allocates local variable space in a new stack frame. A check is made of available stack space. If there isn't room on the stack, a "Not Enough Memory" error is reported and control is transferred to the Main Command Level.

The TRAP #1 routine then executes a LINK instruction. The LINK instruction is explained in detail in the MC68000, MC68020, and MC68030 manuals, and in the Compiler chapter under "How Pascal Programs Use the Stack".

The following illustration shows the stack before the function "part_add" gets its local variable space.

Before the LINK:

```
                    ┌─────────────────────┐
                    │  FUNCTION RESULT    │
          12(SP)→   ├─────────────────────┤
                    │    VALUE of x       │
           8(SP)→   ├─────────────────────┤
                    │    VALUE of y       │
           4(SP)→   ├─────────────────────┤
                    │   RETURN ADDRESS    │
            (SP)→   └─────────────────────┘
```

After the LINK:

```
                    ┌─────────────────────┐
                    │   FUNCTION RESULT    │
         16(A6)──►  ├─────────────────────┤
                    │     VALUE of x       │
         12(A6)──►  ├─────────────────────┤
                    │     VALUE of y       │
          8(A6)──►  ├─────────────────────┤
                    │   RETURN ADDRESS     │
          4(A6)──►  ├─────────────────────┤
                    │      OLD (A6)        │
           (A6)──►  ├─────────────────────┤
                    │        Temp          │
         -4(A6)──►  └─────────────────────┘ ◄─(SP)
```
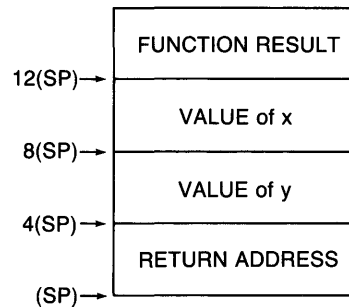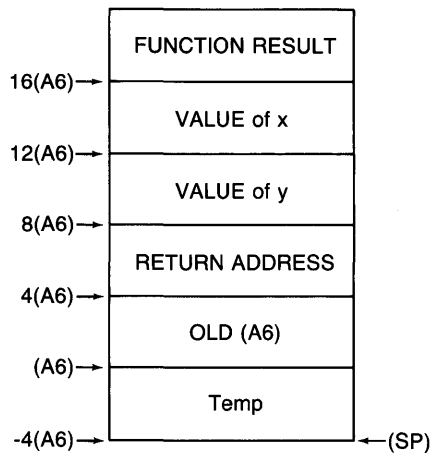
Parameters are now referenced relative to A6 instead of SP. Local variables are referenced at negative offsets from A6.

Local variable space is also needed for copies of some value parameters. As was discussed in the parameter section, value parameters which are larger than 4 bytes have their address put on the stack in place of the value. In order not to alter the value of the actual parameter, a copy must be made in local variable space. Allocate the space using the TRAP instruction, then immediately move the values of the value parameters into the local variable space. This is the case with the parameters to "Procedure Add".

```
                    ┌─────────────────────┐
                    │    ADDRESS of a      │
         16(A6)──►  ├─────────────────────┤
                    │    ADDRESS of b      │
         12(A6)──►  ├─────────────────────┤
                    │   ADDRESS of OUT     │
          8(A6)──►  ├─────────────────────┤
                    │   RETURN ADDRESS     │
          4(A6)──►  ├─────────────────────┤
                    │      OLD (A6)        │
           (A6)──►  ├─────────────────────┤
                    │    COPY of b.i2      │
         -4(A6)──►  ├─────────────────────┤
                    │    COPY of b.i1      │
         -8(A6)──►  ├─────────────────────┤
                    │    COPY of a.i2      │
        -12(A6)──►  ├─────────────────────┤
                    │    COPY of a.i1      │
        -16(A6)──►  └─────────────────────┘ ◄─(SP)
```
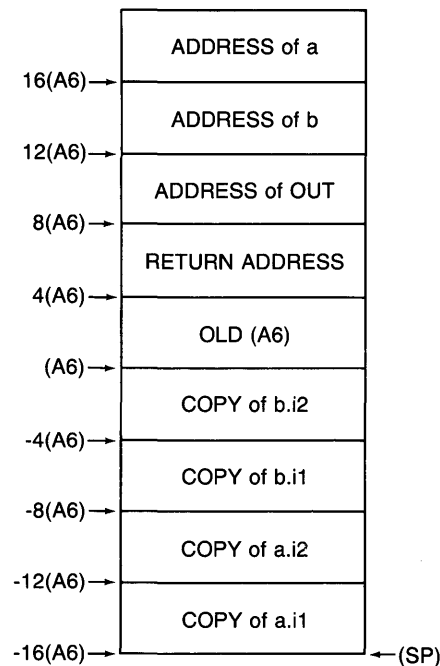
This mapping was accomplished by the following block of code:

```
76    504F              addq.w #8,sp
78    4ED0              jmp (a0)
80    0000              dc.w 0          or dc.b 0,0          or dc.b '  '
- - - - - - - - - - - - - - - - - - - - - - - -   SIMPLE_ADD
82    4E41 FFF0         trap #1,#-16
86    206E 0010         movea.l 16(a6),a0
90    2D58 FFF0         move.l (a0)+,-16(a6)
94    2D50 FFF4         move.l (a0),-12(a6)
98    206E 000C         movea.l 12(a6),a0
102   2D58 FFF8         move.l (a0)+,-8(a6)
106   2D50 FFFC         move.l (a0),-4(a6)
110   2F2D FFF6         move.l SYSGLOBALS-10(a5),-(sp)
114   2F0E              move.l a6,-(sp)
116   487A 005A         pea Rbase+208
120   2B4F FFF6         move.l sp,SYSGLOBALS-10(a5)
```

Module Initialization

Finally, it is necessary to include a module initialization body within each module. The initialization body is a routine which is named by appending the module name to itself, separated by
" _ ".

The purpose of module initialization is to allow for file initialization within the module. Even if a module has no files, the Compiler emits a call to the module initialization body for every module imported into a program. It can be a null routine such as an RTS with the label tacked onto the end of the assembly:

```
simple2_simple2      rts
```

The name of the module initialization body must be marked as an entry point along with the other procedure names using DEF.

Error Recovery

The TRY-RECOVER escape mechanism can be written into assembly language routines for graceful termination of programs that generate errors. TRY-RECOVER is explained in detail in the "Error Trapping and Simulation" chapter.

The section of code that could cause the error is enclosed within the TRY section. The TRY section creates a RECOVER-record on the stack. The record contains the location of the previous RECOVER-record, the stack frame pointer, (A6), and the location of the RECOVER code. The location of this record is saved in a special location that the system knows about. This location is at an offset of -10 in "SYSGLOBALS" (operating SYStem GLOBALS). "SYSGLOBALS" is relative to A5.

An example of the TRY action is taken from the disassembly:

```
 86    206E 0010          movea.l 16(a6),a0
 90    2D58 FFF0          move.l  (a0)+,-16(a6)
 94    2D50 FFF4          move.l  (a0),-12(a6)
 98    206E 000C          movea.l 12(a6),a0
102    2D58 FFF8          move.l  (a0)+,-8(a6)
106    2D50 FFFC          move.l  (a0),-4(a6)
110    2F2D FFF6          move.l  SYSGLOBALS-10(a5),-(sp)
114    2F0E              move.l  a6,-(sp)
116    487A 005A          pea   Rbase+208
120    2B4F FFF6          move.l  sp,SYSGLOBALS-10(a5)
124    598F              subq.l  #4,sp
126    2F2E FFF0          move.l  -16(a6),-(sp)
130    2F2E FFF8          move.l  -8(a6),-(sp)
134    4EBA FF98          jsr   Rbase+32
138    2B5F FFF8          move.l  (sp)+,Gbase-8(a5)
```

After the above code has been written, write the code body of the routine.

The last piece of code must restore the pointer to the previous RECOVER-record and remove the current one from the stack. Control is then transferred to the instruction following the RECOVER section. For example:

```
178    4E76              trapv
180    206E 0008          movea.l 8(a6),a0
184    4CAD 1E00          movem.w Gbase-8(a5),a1-a4
       FFF8
190    4890 1E00          movem.w a1-a4,(a0)
194    2B6F 0008          move.l  8(sp),SYSGLOBALS-10(a5)
       FFF6
200    DEFC 000C          adda.w  #12,sp
204    4EFA 0024          jmp   Rbase+242
208    2C5F              movea.l (sp)+,a6
210    2B5F FFF6          move.l  (sp)+,SYSGLOBALS-10(a5)
214    7064              moveq   #100,d0
216    B06D FFFE          cmp.w   SYSGLOBALS-2(a5),d0
220    6600 0012          bne   Rbase+240
224    4CBA 0F00          movem.w Rbase,a0-a3
```

If an error or exception does occur, the system stores the number of the error in a location at "Sysglobals-2(A5)" and looks at "Sysglobals-10(A5)" to find the location of the RECOVER-record. This location is loaded into the Stack Pointer register (SP). The location of the RE-COVER routine is then popped off the stack and control is transferred to the RECOVER routine. The next value popped off the stack is the stack frame pointer for the RECOVER routine. It is moved to A6. Then the higher level RECOVER-record pointer is popped off the stack and moved to "Sysglobals-10(A5)".

Once these values have been restored, you may examine the value at "Sysglobals-2(A5)" and determine what action to take. If you want to handle the error, you may do so. If not, execute a "TRAP #10" and the problem will ripple out to be handled by the higher level RECOVER routine.

Here is the assembly version of the RECOVER routine:

```
204    4EFA 0024         jmp Rbase+242
208    2C5F              movea.l (sp)+,a6
210    2B5F FFF6         move.l (sp)+,SYSGLOBALS-10(a5)
214    7064              moveq #100,d0
216    B06D FFFE         cmp.w SYSGLOBALS-2(a5),d0
220    6600 0012         bne Rbase+240
224    4CBA 0F00         movem.w Rbase,a0-a3
       FF1C
230    48AD 0F00         movem.w a0-a3,Gbase-8(a5)
       FFF8
236    6000 0004         bra Rbase+242
240    4E4A              trap #10
242    4E5E              unlk a6
244    205F              movea.l (sp)+,a0
246    DEFC 000C         adda.w #12,sp
```

Exception Coding

In your TRY block you may wish to raise certain exception conditions and handle them in the
RECOVER section. This corresponds to the Pascal standard procedure, ESCAPE. When the
condition is determined, store a 16-bit integer value representing the error in "SYSGLOBALS-
2(A5)" and execute a TRAP #10. For example:

```
32     4E41 FFFC         trap #1,#-4
36     202E 000C         move.l 12(a6),d0
40     D0AE 0008         add.l 8(a6),d0
44     4E76              trapv
46     2D40 FFFC         move.l d0,-4(a6)
50     4AAE FFFC         tst.l -4(a6)
54     6C00 000A         bge Rbase+66
58     3B7C 0064         move.w #100,SYSGLOBALS-2(a5)
       FFFE
64     4E4A              trap #10
66     2D6E FFFC         move.l -4(a6),16(a6)
       0010
72     4E5E              unlk a6
74     205F              movea.l (sp)+,a0
76     504F              addq.w #8,sp
```

In your recovery section, check "SYSGLOBALS-2(A5)" to see if you recognize the value. If
you do, make the appropriate recovery. Otherwise, your RECOVER section restores the old
RECOVER-record location and issues another TRAP #10. Thus the error is passed on to the
next RECOVER block.

Returning to Pascal

When returning to Pascal from assembly, the stack must be cleaned up, a function value must
be left on the top of the stack if appropriate, and all Pascal dedicated registers must be restored
(A5, A6 and A7).

You can return to Pascal by leaving the return address on the top of the stack and executing an RTS, or you can store the return address in an address register and execute a JMP indirect through the address register.

```
216    B06D FFFE         cmp.w SYSGLOBALS-2(a5),d0
220    6600 0012         bne Rbase+240
224    4CBA 0F00         movem.w Rbase,a0-a3
       FF1C
230    48AD 0F00         movem.w a0-a3,Gbase-8(a5)
       FFF8
236    6000 0004         bra Rbase+242
240    4E4A              trap #10
242    4E5E              unlk a6
244    205F              movea.l (sp)+,a0
246    DEFC 000C         adda.w #12,sp
250    4ED0              jmp (a0)
252    4E75              dc.w 20085    or dc.b 78,117    or dc.b 'Nu'
```

Declaring External Procedures

Most of the subjects that have been covered in this section are relevant to EXTERNAL procedures.

If you just want to write a routine, put it in the current System Library file (default is the "LIBRARY" file), and call it from Pascal by declaring it as EXTERNAL, you won't need to be concerned with IMPORT text.

You will need to generate EXT and DEF tables. And you will have to deal with parameters. You may or may not want to deal with local variable space. If you want local space, you will reference your parameters relative to (A6). Otherwise, reference them relative to (SP). You will have to write a module initialization body.

The TRY-RECOVER mechanism is also optional. There's always a RECOVER routine somewhere that has to handle those errors. The Operating System puts one around your program before execution.

You must be concerned with the stack. All the parameters must be removed. It must be left in the condition it was in before the calling procedure started preparing for the call.

You must be concerned with restoring A5 and A6 to their original values.

Write the routine, assemble it, and use the Librarian to put it in the System Library. From Pascal, declare it as EXTERNAL, and call it just as if it were a Pascal procedure.

Just remember - if you're not using standard types, every program that calls this routine will have to define the special types just as you had originally defined them.

Instruction Syntax

This section provides details of the syntax of assembly language instructions required by the Workstation Assembler.

Beginning with system version 3.1, the Workstation Assembler supports most of the MC68020 (and with version 3.22, the MC68030) processor and MC68881 (and with version 3.22, the MC68882) co-processor instructions. Thus, you can assemble programs that contain instructions which may not be executable by your computer. Consult the target computer's hardware documentation to determine which processor is installed in it. The few unsupported MC68020/MC68030 opcodes, which are not usually required for user level assembly programs, are:

FGEN   Pass command work to co-processor

PFLUSH Memory Management Unit (MMU) instructions

PLOAD   Memory Management Unit (MMU) instructions

PMOVE   Memory Management Unit (MMU) instructions

PTEST   Memory Management Unit (MMU) instructions

General Syntax

Here is the syntax required for Workstation Assembler instructions. Each portion of the diagram is further described or expanded in the following paragraphs.

Program Line Syntax
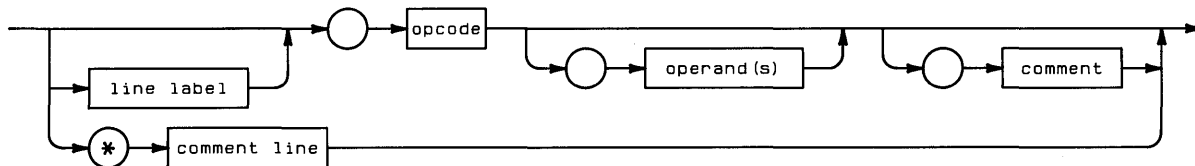
Empty circles denote required spaces. Line labels and comment lines must begin in column 1.

Examples

Labels	Opcodes	Operands	Comments
	RTS		
Label_9	STOP		
	JSR	Subr_name	
Sum	ADD	D1,D2	This is a comment.
* This whole line is a comment.			

Instruction Fields

There are no fixed-width fields within instructions; instead, spaces characters delimit the fields of instructions. For instance, the first space after a line label separates it from the subsequent opcode; the first space after the opcode separates it from the operand; and so forth. Therefore, the following two program lines are equivalent:

```
Label    ADD    D1,D2           This is a comment.

Label ADD D1,D2 This is a comment.
```

This rule dictates that spaces are not permitted within the label, opcode, or operand fields, because the first space encountered after the start of the field ends that field. Instructions are otherwise free-format with respect to spaces; for example, comment fields may have any number of spaces, within the limits of the line width.

Letter Case

Upper- and lower-case characters may be used interchangeably, except inside of literal (quoted) strings.

```
add    d1,d2
MOVE   D2,(A1)
BTst   #31,D2

dc     'This is a literal.'    These two literals are not equivalent.
dc     'THIS IS A LITERAL.'
```

Line Labels
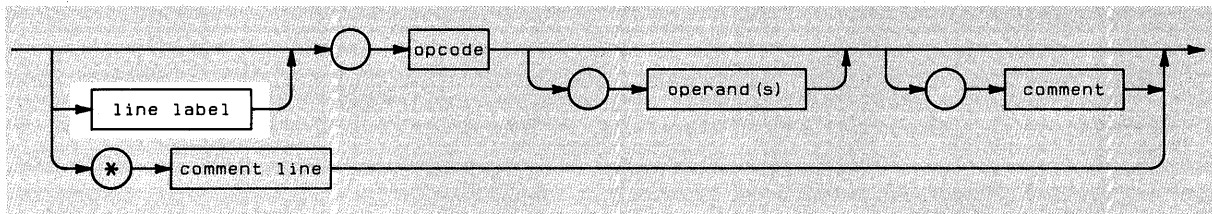
If a line label is present, it must start in column 1 of the line. The opcode must start in column 2 or later (or it will erroneously be considered to be a label).

```
12345678901234567890123456789012345678901234567890

Label  MOVE   A1,A2   Comment field.
       MOVE A1,A2     Comment field.
```

Line labels are in a class of objects called *symbols*, which are described in the "Symbols" discussion of the subsequent "Operands" section.

Opcodes
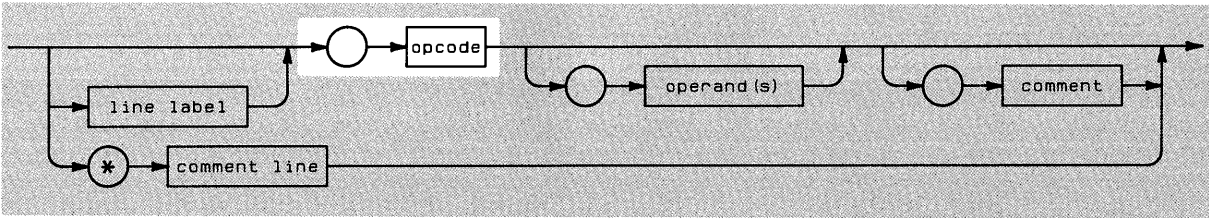
Opcodes (operation codes) are mnemonic abbreviations used for specifying machine language instructions. Here are some examples:

```
ADD
JSR
MOVE
```

The term "opcodes" includes these three types of codes:

- Processor opcodes

- Co-processor opcodes

- Assembler pseudo-opcodes (or simply "pseudo-ops")

Processor and co-processor opcodes are described in this section. Assembler pseudo-ops are described near the end of the chapter.

Processor Opcodes

Most opcodes described in the MC68000, MC68020 and MC68030 user's manuals are supported by the Workstation Assembler. Since the syntax and semantics of each instruction are fully described in those manuals, they will not be described here. The unsupported opcodes are listed at the beginning of this "Instruction Syntax" section.

Note that some instructions have Address, BCD, Immediate, or Quick forms. When possible, Motorola's assembler automatically generates machine instructions for these opcodes (for optimization). However, the Workstation Assembler does not, unless explicitly told to do so.

Co-processor Opcodes

Co-processor opcodes are supported by the Workstation Assembler. The MC68020/MC68030 *User's Manuals* contain a general description of the syntax and semantics of each. Details of particular co-processors, such as the MC68881/MC68882, are in the documentation for the corresponding products. See the "MC68881 and MC68882 Floating Point Co-processor Support" section for a description of exceptions to the syntax and support suggested in the co-processor manuals.

Size Suffixes

Size suffixes can be appended to both of the following items:

- Opcodes and Pseudo-opcodes
- Index registers (in operands)

They specify the size of operand(s) used by the instruction.

The available size suffixes and their definitions are as follows:

Suffix	Meaning	Data Size	Where Used
B	Byte	8 bits	Opcodes only
S	Short	8 bits	Opcodes (Branch instructions only)
W	Word	16 bits	Opcodes and index registers
L	Long	32 bits	Opcodes and index registers

Floating-point size suffixes:

Suffix	Meaning	Data Size	Where Used
S	Single	32 bits	68881/68882 opcodes, DS and the dc.d pseudo-op
D	Double	64 bits	68881/68882 opcodes, DS and the dc.d pseudo-op
X	Extended	80 bits	68881/68882 opcodes, DS and the dc.d pseudo-op
P	Packed	96 bits	68881/68882 opcodes, DS and the dc.d pseudo-op

All instructions that can operate on more than one data size will assume the default size of word (16 bits) unless a size suffix is explicitly specified.

Here are some examples of using these suffixes:

```
ADD.B   D1,D2       ADD registers as Bytes
ADD.W   D1,D2       ADD registers as Words
ADD     D1,D2       ADD registers as Words (default operand length)
ADD.L   D1,D2       ADD registers as Long words

ADD   (A1,A2.W),D1   16 bits of A2 are used as index (default)
ADD   (A1,D2.L),D1   32 bits of D2 are used as index

BEQ.S  Label_Z   Branch on EQual; destination specified by  8 bits
BEQ.W  Label_Z   Branch on EQual; destination specified by 16 bits (always)
BEQ    Label_Z   Branch on EQual; destination specified by 8 or 16 bits
BEQ.L  Label_Z   Branch on EQual; destination specified by 32 bits
```

Note that in the BEQ instruction with no suffix, the Assembler determines whether it will use 8 or 16 bits to specify the destination. In contrast to this, the BEQ.W opcode specifies that 16 bits will be used, even though the destination might fit in 8 bits.

Operands
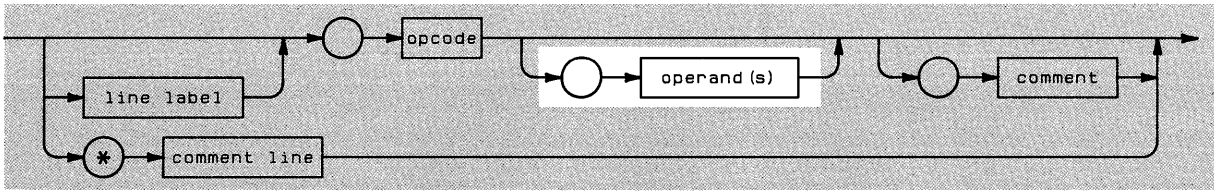
Operands specify the data upon which the operation is to be performed. They are composed of any of the following:

- Constants

- Symbols

- Expressions containing constants, symbols, and operators

- An instance of an addressing mode (see the subsequent "Addressing Modes" section for further information)

Constants

Constants are sequences of ASCII characters that define a numeric value. There are basically four types of constants:

- Decimal numeric constants.

   ```
   123
   2147000111
   ```

   Note that decimal numeric constants may contain any decimal digits 0 through 9; however, they may **not** contain a sign character (+ or -), which would make them an *expression* (see the subsequent "Expressions" section).

- Hexadecimal numeric constants.

   ```
   $19
   $FF20
   ```

   A $ preceding a numeric constant indicates that it is a hexadecimal (base 16) quantity.

- Floating-point numeric constants.

   ```
   3.14
   5.01E9
   -3.14E-99
   2.718E-231
   ```

   The syntax requirements for these constants are the same as for Pascal floating-point (real) constants. See the "Real and Longreal Literals" section of the "Numbers" entry in the *HP Pascal Language Reference manual* for details.

- Literal character constants.

```
'a'
'-.z1'
```

When the characters are enclosed within single quotes, the Assembler emits a series of bytes, one per character, each of which contains the ASCII code of the corresponding character.

Literal character constants may contain 0 to 4 characters. If the single quote character (') is to be part of the constant, you must put two quotes in the literal.

```
'a''b'        Literal is a'b
```

Symbols

Symbols are *names* used in place of values or registers. Symbols must begin with an alphabetic character, but they may subsequently contain digits (0..9), @, $ and _ as well as alphabetic characters. Here are some examples:

```
Symbol
SYM_2
MAIN__main
Z@_$13
```

Symbols may contain any number of characters. The only restriction is that each instruction must be contained entirely on one line. Note also that upper-case and lower-case letters are considered equivalent in symbols; i.e., the symbol MAIN is equivalent to the symbol main.

Here are examples of using symbols in instructions:

```
Symbol1    EQU    RBase+32
           JSR    Symbol1
           ADD    (Symbol2,A1),D1
```

The Location Counter Symbol

The * character is a symbol that signifies the value of the Assembler's location counter (except when the * is in column 1, which indicates a comment line). Here is an example of using the symbol in a branch instruction.

```
           BRA    *-2
```

The location counter points to the memory address at which the instruction begins, and thus is analogous to the processor's program counter (PC). In fact, * is equal to the program counter at the point the instruction is fetched; however, the PC varies from * by 2 or 4 bytes at the point the operands are fetched.

Symbol Types

Here are descriptions of the various types of symbols known to the Assembler:

- **Pre-defined register symbols** are any of the following:

Symbol	Register Specified
AO thru A7	Address registers 0 thru 7
CCR	Condition Code Register
CACR	CAche Control Register
CAAR	CAche Address Register
DO thru D7	Data registers 0 thru 7
DFC	Destination Function Code register
FPO thru FP7	Floating-Point co-processor registers 0 thru 7
FPCONTROL	Floating-Point co-processor control register
FPSTATUS	Floating-Point co-processor status register
FPIADDR	Floating-Point co-processor Immediate ADDRess register
ISP	Interrupt Stack Pointer (A7´)
MSP	Master Stack Pointer (A7″)
SFC	Source Function Code register
SP	Stack Pointer (same as USP and A7)
SR	Status Register
USP	User Stack Pointer (same as SP and A7)
VBR	Vector Base Register

Note that the symbols FPCONTROL, FPSTATUS, and FPIADDR are deviations from Motorola's assembler symbol names; however, you can define the Motorola register symbols as shown in the third example below.

- **User-defined register symbols** are created with the EQU (equate) pseudo-op:

```
StackPointer   EQU   SP
MyAddressReg   EQU   A1
STATUS         EQU   FPSTATUS
```

Note that these are the only type of symbols that need to be defined *before* they are used.

- **Absolute symbols** are those which either follow an `ORG` pseudo-op or are equated to an absolute expression. Here are some examples:

```
        ORG   $FFFFF000
AbsSym1 EQU   $FFFFFED2
AbsSym2 EQU   AbsSym1+16
AbsSym3 EQU   RelSym1-RelSym2  Note: DIFFERENCE of 2 relatives is absolute
```

- **Relative symbols** are those which follow a `RORG` pseudo-op or are equated to a relative expression.

```
        RORG  *                 Relative to Program Counter (PC)
RelSym1 EQU   RBase+16
RelSym2 EQU   RelSym1
RelSym3 EQU   RelSym2+AbsSym2   Note: CANNOT add 2 relatives
```

- **External symbols** are those which are defined in another module (by a `DEF` Assembler pseudo-op, or by another language's compiler). They can be either absolute or relative. Here examples of how external symbols are defined (in the module to which they are external):

```
REFA  AbsExtSym1    Absolute external symbol.
REFR  RelExtSym1    Relative external symbol.
```

Expressions

Expressions are the general case of operand: they may be just symbols; or they can be more complex combinations of symbols, constants, and operators. The operators in expressions are limited to the following:

- \- subtraction, or unary minus

- \+ addition, or unary plus

- ! bit-wise logical OR

- & bit-wise logical AND
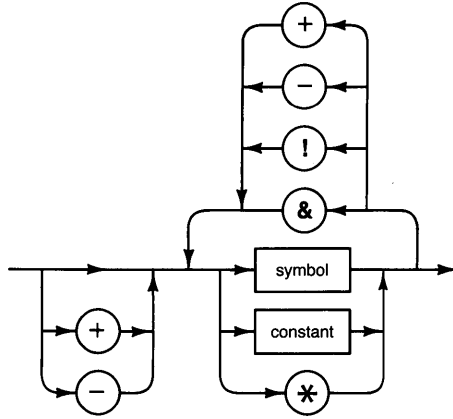
Expressions are evaluated in strict left-to-right order, and parentheses are *not* allowed. Only *one* external symbol, or symbol equated to an expression containing an external symbol, is allowed per expression. Also note that you cannot add two relative symbols (although you can subtract two, since the difference is an absolute value). Expressions cannot have real number operands.

```
2+2

RelSym1+48+$DF00

RelSym1-RelSym2+RelSym3+AbsExtSym1

AbsSym1-AbsSym2+RelSym3+AbsSym3
```

Addressing Modes

The Workstation Assembler supports all of the addressing modes of the current 68000 family of processors; this section describes the syntax required to access each mode.

With system version 3.1, the Workstation Assembler was updated to support the 68020 processor's addressing modes. (The 68030 processor was supported beginning with the 3.22 release, and its addressing modes are the same as 68020 modes.) Since these new modes cannot be accessed with the old address syntax, new syntax is required. However, *note that the old 68000 syntax is still supported in all instances.*

The following table shows the syntax and operand components of all supported addressing modes. Descriptions of operand components are given in the legend on the next page.

Description of Mode	68000 & 68010	68020/68030
Data Register Direct	Dn	same
Address Register Direct	An	same
Address Register Indirect	(An)	same
Address Register Indirect with Post-increment	(An)+	same
Address Register Indirect with Pre-decrement	-(An)	same
Address Register Indirect (absolute displacement)	d16(An)	(bd,An)
Address Register Indirect with Index (absolute displacement)	d8(An,Rn.SIZE)	(bd,An,Xn.SIZE*SCALE)
Memory Indirect Post-indexed	n/a	([bd,An],Xn.SIZE*SCALE,od)
Memory Indirect Pre-indexed	n/a	([bd,An,Xn.SIZE*SCALE],od)
PC Memory Indirect Post-indexed	n/a	([rbd],Xn.SIZE*SCALE,od)
PC Memory Indirect Pre-indexed	n/a	([rbd,Xn.SIZE*SCALE],od)
Absolute Address	Expr	same
Program Counter Indirect (relative displacement)	rd16	rbd
Program Counter Indirect with Index (relative displacement)	rd8(Xn.SIZE)	(rbd,Xn.SIZE*SCALE)
Immediate Data	#Expr	same
Register List (MOVEM)	Ai-Aj/Dm-Dn	same
Bit Field Specifier	n/a	operand{offset:width}

An	= Address register symbol: A0 thru A7
Dn	= Data register symbol: D0 thru D7
Xn	= Index register symbol: A0 thru A7, or D0 thru D7
Rn	= Equivalent to Xn (see above)
d8	= absolute expression (8-bit): -2^7 thru $2^7 -1$
d16	= absolute expression (16-bit): -2^{15} thru $2^{15}-1$
bd	= absolute expression (32-bit): -2^{31} thru $2^{31}-1$
od	= absolute expression (32-bit): -2^{31} thru $2^{31}-1$
rd8	= relative expression (8-bit): $PC-2^7$ thru $PC+2^7 -1$
rd16	= relative expression (16-bit): $PC-2^{15}$ thru $PC+2^{15}-1$
rbd	= relative expression (32-bit): $PC-2^{31}$ thru $PC+2^{31}-1$
SIZE	= literal: W (or no suffix) specifies Word operand
	= literal: L specifies Long word operand
SCALE	
	= literal: 1
	2
	4
	8
n/a	= not implemented in the older "68000" Workstation Assembler
Expr	= expression (relative or absolute)
Ai-Aj/Dm-Dn	= List of registers: - means "thru"; / means "and". The values of i, j, m, and n can range from 0 thru 7. Examples: A0-A3/D1-D4/A5, and D5-D7/D0/A0
operand	= any of the operands allowed with instructions that can operate on "bit fields".
offset	= literal or symbol that specifies starting bit of a bit field.
width	= literal or symbol that specifies number of bits in a bit field. Examples: D7{2:3}, and (A0){D1:D2}

Operand Components: Order and Optionality

In the above table, an operand is the whole quantity shown in one column entry. Here are some examples of single operands:

```
AbsExpr1(A1,A2.L)
(AbsExpr2,A1,D3.W*4)
([AbsExpr3,A4],A3.L,AbsExpr4)
```

Thus, some of the components of these operands are AbsExpr1, A1, A2.L, AbsExpr2, D3.W*4, and
so forth.

The newer Workstation Assembler allows you to vary the order of these operand components
(however, *only* with the new 68020 syntax). Here are some examples of varying the order of
operand components (note that all these operands are equivalent):

    ([AbsExpr3,A4],A3.L,AbsExpr4)
    ([A4,AbsExpr3],A3.L,AbsExpr4)
    ([A4,AbsExpr3],AbsExpr4,A3.L)
    (AbsExpr4,A3.L,[A4,AbsExpr3])

The Workstation Assembler also allows you optionally to omit some of the operand components,
(however, only with the new 68020/68030 syntax) as shown in the following examples (these
operands are *not* equivalent):

    ([AbsExpr1,A1],A5.L,AbsExpr2)
    ([A1],A5.L,AbsExpr2)
    ([AbsExpr1],A5.L)
    ([A1])
    (D1)

Note that whenever any operand component is omitted, an effective value of 0 is used for that
component.

Comments and Comment Lines
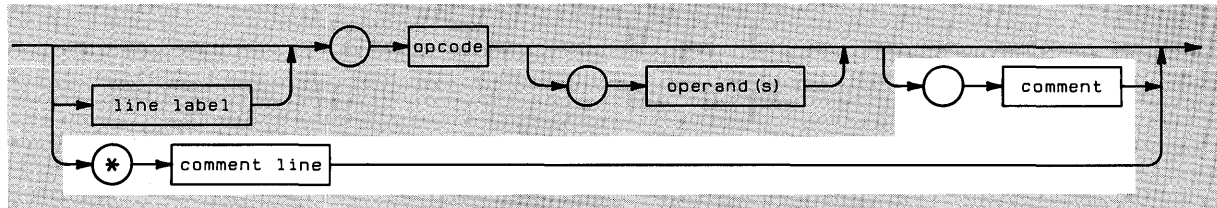
The first space following an operand (or the opcode in an instruction with no operands)
terminates it; the remainder of the characters on the line, if any, are regarded as comments.

    Label ADD D1,D2 This is a comment.
           RTS This is also a comment (since RTS has no operands).

An asterisk (*) in column 1 indicates line that the entire line is a comment; therefore, any
instructions on the line will be ignored.

    12345678901234567890123456789012345667890

    * These are comment lines.
    * Add word addressed by A1 to the value of the error counter.
    *

MC68881 and MC68882 Floating Point Co-processor Support

This section describes the Motorola MC68881/MC68882 floating point co-processor support provided by the HP Pascal Workstation Assembler. Note that the support provided by the Assembler is a subset of the co-processor's full capabilities.

Assembler Support of the Co-processor

Below is a list showing which co-processor capabilities are supported and which are unsupported by the Assembler.

- All opcodes are supported.

- All sizes and types of operands are supported, with the exception that floating-point constants and immediate operands are restricted as follows:

    - The floating-point literal constant syntax is that defined in the *HP Pascal Language Reference* under "Numbers", such that a period must be present, and the "L" exponent flag is not allowed (i.e. only the "E" exponent is allowed).

    - Only double-precision constants are allowed (i.e. ".D" size suffix).

    - The size suffix ".D" must explicitly be given.

    - All non-zero constants and immediate operands are normalized (i.e. 0.0 and normalized are the only supported IEEE types).

    - Use of floating-point values in expressions is not supported.

- Assembler Pseudo-op DS allocates space in memory for all types and sizes of operands.

- Assembler Pseudo-op DC, which reserves storage space, only supports double-precision operands (i.e. .D but not .S or .X).

- No pseudo-ops (directives) are provided to control the rounding mode, so the default of "round to nearest integer value" occurs.

- No pseudo-ops (directives) are provided to allow use of different co-processors. Therefore, the default co-processor id of 1 is used, which is the MC68881 or MC68882 co-processor.

Assembler Pseudo-Op Reference

The following is a list of the commands which direct the assembler to take the described actions. For a list of the machine commands, see the *MC68000 User's Manual.*

COM

Used to define a global area.
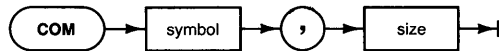
Item	Description	Range
symbol	An identifier for the global area	see "Symbols"
size	A numeric expression	-2^{31} thru $2^{31} -1$

Semantics

The exact location of the global area will be determined at link time. The **symbol** is DEFined as an entry point. The amount of space is specified by the absolute value of the expression. If **size** is negative, the value of the symbol will be the offset from (A5) to the top of the global area and variables will have negative offsets from the symbol. This is how the Compiler does it. If **size** is positive, the symbol's value will be the bottom of the area, relative to (A5), and offsets will be positive. Only one COM statement allowed per assembly.

DC

Used to define some constant value or values, including string literals, into storage.
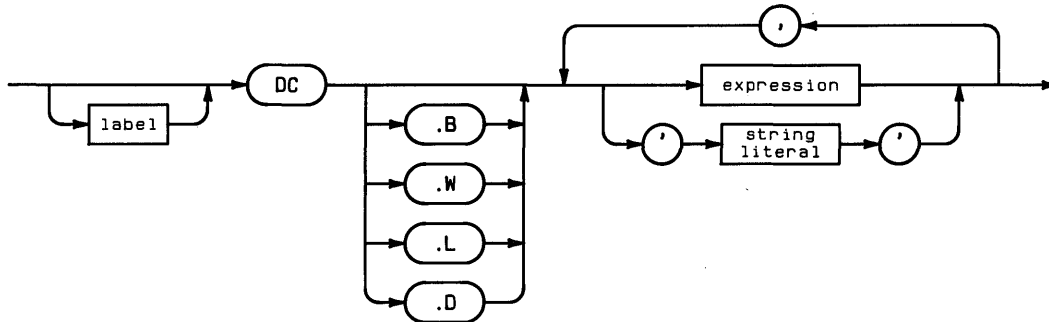
Item	Description	Range
label	An identifier for the constant	see "Symbols"
value	An expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$
string literal	A string of characters	The instruction must be contained on one line

Semantics

Size suffixes may be used to specify the units of storage into which the values will be assigned. In the case of string literals, the amount of storage needed will be determined by the assembler and each character will be assigned into a byte, with the last unit null padded if necessary.

DECIMAL

Causes addresses in the listing to be printed in decimal rather than in Hex notation.

DECIMAL

DEF

Defines a label or list of labels as entry points for other modules.

DEF ─── label
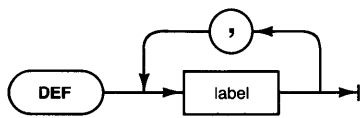
Item	Description	Range
label	An entry point identifier	see "Symbols"

DS

Reserves storage space.

label DS .B .W .L .S .D .X .P expression
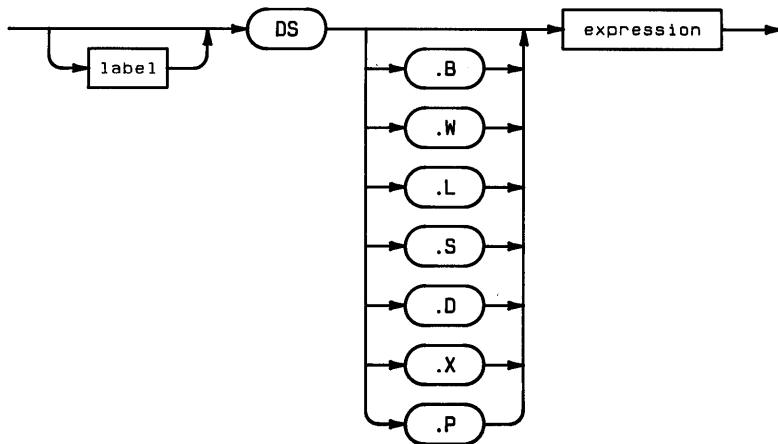
Item	Description	Range
label	the identifier for the data space	see "Symbols"
number	an expression that can be evaluated in pass 1	0 thru $2^{31}-1$

Semantics

The units of space are specified by the size suffix. The number of units is determined by the expression.

The ".S", ".D", ".X", and ".P" size suffixes are supported only for the MC68881/MC68882 floating-point math coprocessor.

END

Indicates the end of the assembly. This should be the last line of the assembly.

EQU

Assigns the value and attribute (absolute or relative) of the expression to the label. Equating a symbol to a register is allowed.

Item	Description	Range
label	the identifier for the data space	see "Symbols"
value	an expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

INCLUDE

Specifies a file to be merged into the assembly at the point where the instruction is located. The '.TEXT' suffix will be automatically appended to the file name. The INCLUDEd file may not contain another INCLUDE.
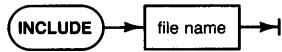
LLEN

Used to specify the column width of your printer.
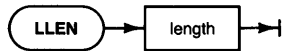
LIST

Turns the printer listing back on. You must have requested a listing when the assembler was initiated. LIST is used with NOLIST to exclude blocks of text from the listing.

LMODE

Specifies a symbol or list of symbols to be accessed using long absolute addressing mode. Overrides short addressing and PC relative mode implications of REFR, ORG, and RORG.
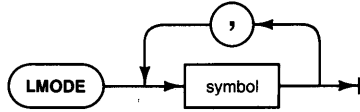
Item	Description	Range
symbol	A location identifier	see "Symbols"

LPRINT

(Default) Causes all output from DC statements to be printed. (See SPRINT)

MNAME

Used to assign a module name to an a assembly. The default is to assign the file name to the module.
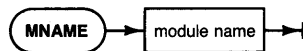
NOLIST

Turns off the listing until a LIST is encountered.

NOOBJ

Requests that no object code be produced.

```
( NOOBJ )──►┤
```

NOSYMS

Inhibits the listing of the symbol table at the end of the program.

```
( NOSYMS )──►┤
```

ORG

Specifies an absolute origin. When used with the ".L" option, it forces long mode addressing for forward and external references. Otherwise short absolute addressing mode is implied.

```
( ORG )──────────►──────►[ absolute  ]──►┤
         └──( .L )──┘       origin
```
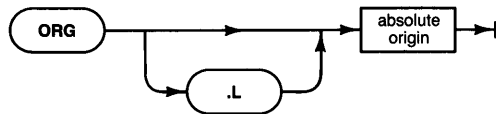
Item	Description	Range
absolute origin	A numeric expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

PAGE

Advances listing to top of next page. This command will not be printed on the listing.

```
( PAGE )──►┤
```

REFA

Defines a symbol or list of symbols as external and absolute references. The size of the effective address is implied by the ORG statement.
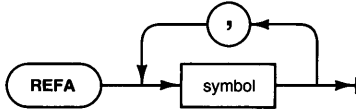
Item	Description	Range
symbol	A location identifier	see "Symbols"

REFR

Defines a symbol or list of symbols as external and PC relative references.
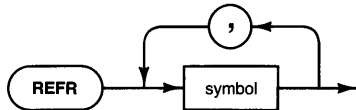
Item	Description	Range
symbol	A location identifier	see "Symbols"

RMODE

Specifies a symbol or list of symbols for access using PC relative addressing. Overrides all other addressing mode specifications.
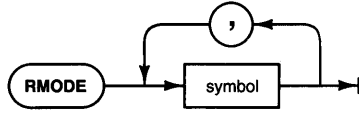
Item	Description	Range
symbol	A location identifier	see "Symbols"

RORG

Sets a relocatable origin. Using the 'L' option, forces long absolute addressing mode for forward and external references. Otherwise PC relative addressing mode is implied for forward references and short absolute addressing mode for REFA symbols.
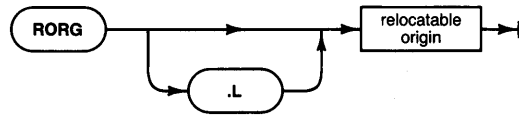
Item	Description	Range
relocatable origin	A numeric expression that can be evaluated in pass 1	-2^{31} thru $2^{31}-1$

SMODE

Specifies a symbol or list of symbols to be accessed using short absolute addressing mode. Overrides all other addressing mode specifications.
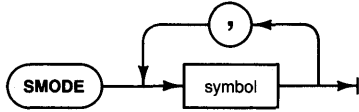
Item	Description	Range
symbol	A location identifier	see "Symbols"

SPC

Directs the assembler to generate the specified number of blank lines. Used to separate blocks of code or blocks of comments on the listing.
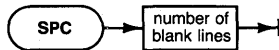
SPRINT

Print only the first line of output for the DC statements. Otherwise each word of memory used to store the constant is printed.

SRC

Used to specify the IMPORT text information which the Compiler needs when importing the module. Use one SRC for each line of IMPORT text. (see programming section)

```
( SRC )──►│ line of export text │──►│
```

START

Specifies a start location for execution of the main program. Use only in the main program.

```
( START )──►│ start location │──►│
```
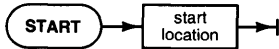
Item	Description	Range
start location	An integer numeric expression	-2^{31} thru $2^{31}-1$

TTL

Specifies a title to appear on each page of the assembler listing.

```
( TTL )──►│ page title │──►│
```
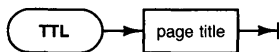
The Examples

Listings of the two programs and two modules are given here and also have been provided on the documentation disc (DOC:). On the disc they are provided in source and object form. The file (ASMB_P1) imports the file (ASMB_M1). These are both Pascal files. The Pascal file (ASMB_P2) imports the assembly language file (ASMB_M2).

If you want to see them work, you must either use the Librarian to link the modules to the programs, P-load the modules, or put the modules in the current System Library. You can then execute the two programs.

The Sample Pascal Programs

This Program Imports the Pascal Module

```
$search '#3:ASMB_M1
Program test(input,output);
Import simple;
var i,j,k : rec;
begin
    initialize;
    i.i1:=1; i.i2:=2;
    j.i1:=3; j.i2:=4;
    add(i,j,k);
    writeln(k.i1,k.i2)
end.
```

This Program Imports the Assembly Module

```
$search '#3:ASMB_M2
Program test(input,output);
Import simple2;
var i,j,k : rec;
begin
    initialize;
    i.i1:=1; i.i2:=2;
    j.i1:=3; j.i2:=4;
    add(i,j,k);
    writeln(k.i1,k.i2)
end.
```

The Sample Pascal Module

```
$sysprog$              (*to enable try-recover*)

module simple;
export
  type
    rec = record
                i1: integer;
                i2: integer;
          end;
  const
    zero = rec [i1:0,i2:0];
  var
    lastresult: rec;

  procedure initialize;
  procedure add (a,b: rec; var out: rec);

implement
  var
    sum: rec;

  procedure initialize;
    begin   sum := zero   end;

  function partadd (x,y: integer): integer;
  var   temp: integer;
  begin
    temp := x+y;
    if temp < 0 then escape(100);
    partadd := temp;
  end; (*partadd*)

  procedure add (a,b: rec; var out: rec);
  begin
    try
      lastresult.i1 := partadd(a.i1,b.i1);
      lastresult.i2 := partadd(a.i2,b.i2);
      sum.i1 := sum.i1+lastresult.i1;
      sum.i2 := sum.i2+lastresult.i2;
      out := lastresult;
    recover
      if escapecode = 100
        then lastresult := zero
        else escape(escapecode);
  end; (*add*)

end.
```

The Disassembly of the Module

```
MODULE     SIMPLE     Created  8-Oct-82
NOTICE:  (none)
  Produced by Pascal Compiler of 20-Sep-82
  Revision number 2
  Directory size    172 bytes
  Module size      3072 bytes
  Module NOT executable
  Code base             0      Size        254 bytes
  Global base           0      Size         16 bytes
  EXT    block   5     Size          20 bytes
  DEF    block   3     Size         114 bytes
  EXPORT block   1     Size         192 bytes
  There are        1 TEXT records


TEXT RECORD #          1  of 'SIMPLE':
  TEXT  start block   2      Size        254 bytes
  REF   start block   4      Size         42 bytes
  LOAD address      Rbase

      0   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
      2   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
      4   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
      6   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
      8   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
- - - - - - - - - - - - - - - - - - - - - SIMPLE_INITIALIZE
     10   4E41 0000        trap #1,#0
     14   4CBA 0F00        movem,w Rbase,a0-a3
          FFEE
     20   48AD 0F00        movem,w a0-a3,Gbase-16(a5)
          FFF0
     26   4E5E             unlk a6
     28   4E75             rts
     30   0000             dc,w 0        or dc,b 0,0        or dc,b '  '


- - - - - - - - - - - - - - - - - - - - - - -
     32   4E41 FFFC        trap #1,#-4
     36   202E 000C        move,l 12(a6),d0
     40   D0AE 0008        add,l 8(a6),d0
     44   4E76             trapv
     46   2D40 FFFC        move,l d0,-4(a6)
     50   4AAE FFFC        tst,l -4(a6)
     54   6C00 000A        bge Rbase+66
     58   3B7C 0064        move,w #100,SYSGLOBALS-2(a5)
          FFFE
     64   4E4A             trap #10
     66   2D6E FFFC        move,l -4(a6),16(a6)
          0010
     72   4E5E             unlk a6
     74   205F             movea,l (sp)+,a0
     76   504F             addq,w #8,sp
     78   4ED0             jmp (a0)
     80   0000             dc,w 0        or dc,b 0,0        or dc,b '  '
```

- SIMPLE_ADD   (

```
 82   4E41 FFF0            trap #1,#-16
 86   206E 0010            movea.l 16(a6),a0
 90   2D58 FFF0            move.l (a0)+,-16(a6)
 94   2D50 FFF4            move.l (a0),-12(a6)
 98   206E 000C            movea.l 12(a6),a0
102   2D58 FFF8            move.l (a0)+,-8(a6)
106   2D50 FFFC            move.l (a0),-4(a6)
110   2F2D FFF6            move.l SYSGLOBALS-10(a5),-(sp)
114   2F0E                 move.l a6,-(sp)
116   487A 005A            pea Rbase+208
120   2B4F FFF6            move.l sp,SYSGLOBALS-10(a5)
124   598F                 subq.l #4,sp
126   2F2E FFF0            move.l -16(a6),-(sp)
130   2F2E FFF8            move.l -8(a6),-(sp)
134   4EBA FF98            jsr Rbase+32
138   2B5F FFF8            move.l (sp)+,Gbase-8(a5)
142   598F                 subq.l #4,sp
144   2F2E FFF4            move.l -12(a6),-(sp)
148   2F2E FFFC            move.l -4(a6),-(sp)
152   4EBA FF86            jsr Rbase+32
156   2B5F FFFC            move.l (sp)+,Gbase-4(a5)
160   202D FFF8            move.l Gbase-8(a5),d0
164   D1AD FFF0            add.l d0,Gbase-16(a5)
168   4E76                 trapv
170   202D FFFC            move.l Gbase-4(a5),d0
174   D1AD FFF4            add.l d0,Gbase-12(a5)
178   4E76                 trapv
180   206E 0008            movea.l 8(a6),a0
184   4CAD 1E00            movem.w Gbase-8(a5),a1-a4
      FFF8
190   4890 1E00            movem.w a1-a4,(a0)
194   2B6F 0008            move.l 8(sp),SYSGLOBALS-10(a5)
      FFF6
200   DEFC 000C            adda.w #12,sp
204   4EFA 0024            jmp Rbase+242
208   2C5F                 movea.l (sp)+,a6
210   2B5F FFF6            move.l (sp)+,SYSGLOBALS-10(a5)
214   7064                 moveq #100,d0
216   B06D FFFE            cmp.w SYSGLOBALS-2(a5),d0
220   6600 0012            bne Rbase+240
224   4CBA 0F00            movem.w Rbase,a0-a3
      FF1C
230   48AD 0F00            movem.w a0-a3,Gbase-8(a5)
      FFF8
236   6000 0004            bra Rbase+242
240   4E4A                 trap #10
242   4E5E                 unlk a6
244   205F                 movea.l (sp)+,a0
246   DEFC 000C            adda.w #12,sp
250   4ED0                 jmp (a0)
252   4E75                 dc.w 20085     or dc.b 78,117     or dc.b 'Nu'
```

The Assembly Language Module

```
            mname  simple2

            src  module  simple2;
            src  export
            src    type
            src        rec = record
            src              i1 : integer;
            src              i2 : integer;
            src        end;
            src    const
            src        zero = rec[i1:0,i2:0];
            src    var
            src        lastresult : rec;
            src    procedure  initialize;
            src    procedure  add(a,b : rec;
            src                   var out : rec);
            src  end;

            com  simple2,-16

            def  simple2_add
            def  simple2_initialize
            def  simple2_zero,simple2_simple2

            refa  sysglobals

lastresult          equ  simple2-8
lastresult_i1       equ  simple2-8
lastresult_i2       equ  simple2-4
sum                 equ  simple2-16  (all are relative to a5)
sum_i1              equ  simple2-16
sum_i2              equ  simple2-12
escapecode          equ  sysglobals-2
recover_rec         equ  sysglobals-10

            rorg  0

simple2_zero        dc.l   0,0


simple2_initialize  trap #1     (stack check)
            dc.w   0      (no local space)

            movem.l  simple2_zero,a0-a1

            movem.l  a0-a1,sum(a5)


            unlk  a6
            rts

simple2_partadd   trap #1
            dc.w   -4

result              equ  16
x                   equ  12
y                   equ  8        (all are relative to a6)
ret_addr            equ  4
```

```
dyn_link          equ 0
temp              equ -4

        move.l  x(a6),d0   (temp:=x+y)
        add.l   y(a6),d0
        trapv      (overflow check)
        move.l  d0,temp(a6)

        tst.l   temp(a6)   (if temp<0)
        bge  past_escape
        move #100,escapecode(a5)

        trap #10      (then escape 100)

past_escape      move.l temp(a6),result(a6)

*                                  (partadd:=temp)

        movea.l ret_addr(a6),a0
        unlk a6
        adda.l  #12,sp

        jmp (a0)

simple2_add       trap #1      (stack check)
        dc.w -16   (for param copies)

a_addr            equ 16
b_addr            equ 12
out_addr          equ 8
ret_addr2         equ 4
dyn_link2         equ 0        (relative to a6)
b_i2_copy         equ -4
b_i1_copy         equ -8
a_i2_copy         equ -12
a_i1_copy         equ -16

        movea.l a_addr(a6),a0     (making local
        move.l  (a0)+,a_i1_copy(a6)   copies)
        move.l  (a0),a_i2_copy(a6)
        movea.l b_addr(a6),a0
        move.l  (a0)+,b_i1_copy(a6)
        move.l  (a0),b_i2_copy(a6)

        move.l  recover_rec(a5),-(sp)    (TRY)
        move.l  a6,-(sp)
        pea     recover_addr
        move.l  sp,recover_rec(a5)

        subq.l  #4,sp   (calling partadd)
        move.l  a_i1_copy(a6),-(sp)
        move.l  b_i1_copy(a6),-(sp)
        jsr  simple2_partadd
        move.l  (sp)+,lastresult_i1(a5)

        subq.l  #4,sp   (calling partadd)
        move.l  a_i2_copy(a6),-(sp)
```

```
        move.l b_i2_copy(a6),-(sp)
        jsr   simple2_partadd
        move.l (sp)+,lastresult_i2(a5)

        move.l lastresult_i1(a5),d0  (sum:=
        add.l  d0,sum_i1(a5)        sum+lastresult)
        trapv
        move.l lastresult_i2(a5),d0
        add.l  d0,sum_i2(a5)
        trapv

        movea.l out_addr(a6),a0
        movem.l lastresult(a5),a1-a2

        movem.l a1-a2,(a0)      (out:=lastresult)

        move.l 8(sp),recover_rec(a5)

        adda.l #12,sp           (end of TRY)

        jmp   past_recover

recover_addr    movea.l (sp)+,a6   (RECOVER)
        move.l (sp)+,recover_rec(a5)
        moveq #100,d0   (if escapecode=100)
        cmp.w  escapecode(a5),d0
        bne   sys_error
        movem.l simple2_zero,a0-a1

*                               (then lastresult:=0)
        movem.l a0-a1,lastresult(a5)

        bra   past_recover
sys_error       trap #10        (else escape)

past_recover    unlk  a6
        movea.l (sp)+,a0
        adda.l #12,sp

        jmp  (a0)

simple2_simple2    rts    (initialization body)

        end


    *** 68000 ASSEMBLER SYMBOL TABLE DUMP ***

            EXTERNAL SYMBOLS

        SYMBOL      TYPE    DEF     VALUE
SIMPLE2         ABS     19      00000001
SYSGLOBALS      ABS     25      00000002

            INTERNAL SYMBOLS

        SYMBOL      TYPE    DEF     EQU SYM         VALUE
A0              AREG    0                       00000000
A1              AREG    0                       00000001
A2              AREG    0                       00000002
A3              AREG    0                       00000003
A4              AREG    0                       00000004
A5              AREG    0                       00000005
A6              AREG    0                       00000006
A7              AREG    0                       00000007
A_ADDR          ABS     80                      00000010
A_I1_COPY       ABS     88                      FFFFFFF0
A_I2_COPY       ABS     87                      FFFFFFF4
B_ADDR          ABS     81                      0000000C
```

```
B_I1_COPY           ABS      86                          FFFFFFF8
B_I2_COPY           ABS      85                          FFFFFFFC
CCR                 STREG     0                          00000005
D0                  DREG      0                          00000000
D1                  DREG      0                          00000001
D2                  DREG      0                          00000002
D3                  DREG      0                          00000003
D4                  DREG      0                          00000004
D5                  DREG      0                          00000005
D6                  DREG      0                          00000006
D7                  DREG      0                          00000007
DYN_LINK            ABS      56                          00000000
DYN_LINK2           ABS      84                          00000000
ESCAPECODE          ABS      33  SYSGLOBALS       +      FFFFFFFE
LASTRESULT          ABS      27  SIMPLE2          +      FFFFFFF8
LASTRESULT_I1       ABS      28  SIMPLE2          +      FFFFFFF8
LASTRESULT_I2       ABS      29  SIMPLE2          +      FFFFFFFC
OUT_ADDR            ABS      82                          00000008
PAST_ESCAPE         REL      69                          0000003E
PAST_RECOVER        REL     140                          000000F4
RECOVER_ADDR        REL     129                          000000D2
RECOVER_REC         ABS      34  SYSGLOBALS       +      FFFFFFF6
RESULT              ABS      52                          00000010
RET_ADDR            ABS      55                          00000004
RET_ADDR2           ABS      83                          00000004
SIMPLE2_ADD         REL      77                          00000052
SIMPLE2_INITIALIZE REL          40                       00000008
SIMPLE2_PARTADD     REL      49                          0000001C
SIMPLE2_SIMPLE2     REL     145                          00000100
SIMPLE2_ZERO        REL      38                          00000000
SP                  AREG      0                          00000007
SR                  STREG     0                          00000006
SUM                 ABS      30  SIMPLE2          +      FFFFFFF0
SUM_I1              ABS      31  SIMPLE2          +      FFFFFFF0
SUM_I2              ABS      32  SIMPLE2          +      FFFFFFF4
SYS_ERROR           REL     138                          000000F2
TEMP                ABS      57                          FFFFFFFC
USP                 STREG     0                          00000007
X                   ABS      53                          0000000C
Y                   ABS      54                          00000008
```

The Librarian

<div style="text-align: right; font-size: 2em; font-weight: bold;">8</div>

Introduction

It may seem obvious that the Librarian's purpose is to manage libraries. However, all the things that it can do to fulfill this responsibility may not be as obvious. This chapter will help to put all of the Librarian's capabilities into perspective. The chapter first describes libraries and object modules, providing some relevant background information that will help you to understand the Librarian operations described in the latter sections of the chapter.

Here is a brief overview of the operations you can perform with the Librarian:

- Add object modules to or remove them from libraries. For instance, you can add object modules to the System Library so that the modules will be found and loaded automatically when any program that imports them is loaded for execution.

- Link the directories of the object modules in a library file. This operation reduces the file's size.

- Obtain detailed information about the object modules in a library file. For instance, you can unassemble a compiled Pascal object file and get the Assembler language object code. The Librarian can disassemble all instructions for the MC68000 family of processors, as well as MC68881/MC68882 math co-processor instructions.

- Create new system Boot files. This operation is used to create files that are found and loaded by the Boot ROM and in turn load a system.

Let's look more closely at library files, what is in them, and how to use them.

Prerequisites

This chapter presents simple examples of user modules and libraries. If you find that you want more information about modules as you read this chapter, read the sections of the Compiler and Assembler chapters that describe modules.

If you are going to be using the Librarian for purposes other than adding modules to and removing them from the System Library (usually LIBRARY) or Initialization Library (BOOT:INITLIB), then you should also be familiar with the concepts presented in the Assembler chapter.

Library Overview

This section presents some important terms and concepts you will need to know in order to understand libraries. It will help you see when and why you will need to use the Librarian.

Modules and Libraries

Libraries are *object files*. They contain zero or more *object modules*. Object modules are the product of the Compiler or Assembler[1]. For instance, compiling a Pascal source module generates an object module which is placed in an object file. This file is actually a library, because it contains an object module.

An object file is composed of a *directory* of the module(s) that it contains, followed by the object modules themselves. Here is a pictorial representation of an object file.

OBJECT FILE

Library Directory
Object Module Directory
Define Source
Ext Table
Def Table
Text Record
Ref Tables

} Object Module

•
•
•

Text Record
Ref Tables
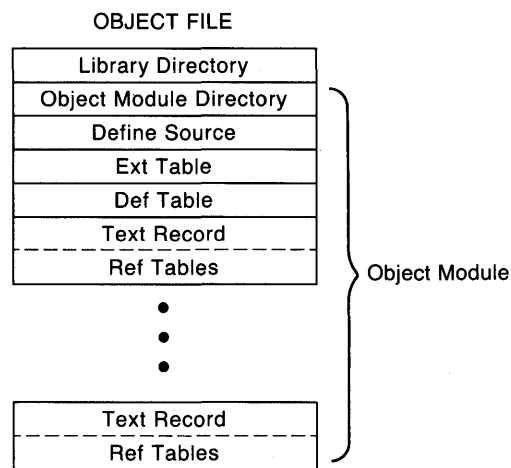
The terms Define Source, Ext Table, and so forth are defined in the Glossary of Object Code Terminology at the end of the chapter.

---

[1] Complete descriptions of how to produce and use Pascal and Assembler modules are provided in the Compiler and Assembler chapters.

What the Librarian Does

The Librarian's purpose is to manage object modules. The Librarian can also produce object files; however, these files consist of object modules produced by the Compiler or Assembler. It can create library files and add modules to them or remove modules from them. The intent of these libraries is to provide a convenient location to store object modules. The following drawing shows the relationship of object modules in an object file (library):

OBJECT FILE

Library Directory		
Object Module Directory		Object Module Directory
Define Source		Define Source
Ext Table	• • •	Ext Table
Def Table		Def Table
Text Record		Text Record
Ref Tables		Ref Tables
•		•
•		•
•		•
Text Record		Text Record
Ref Tables		Ref Tables
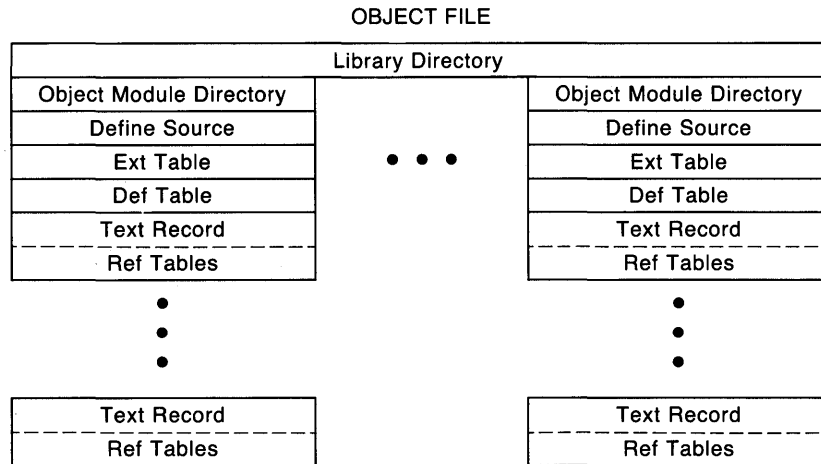
Example Modules

For this example, we will be using three example library modules provided on the DOC: disc shipped with your system. One contains a compiled program (PROG_1.CODE), and the other two contain compiled modules (MOD_2.CODE and MOD_3.CODE).

The DOC: disc also contains the source versions of these modules. Although this chapter will only be dealing specifically with the object versions, it is a good learning experience to compile the source versions to see how the Compiler deals with imported modules. One method is briefly outlined in the next section.

Here are source listings and brief explanations of each of the example modules.

Source Listing of PROG_1.CODE

```
PROGRAM ProgramOne(OUTPUT);

IMPORT ModuleTwo;

BEGIN
    WRITELN;
    WRITELN;
    WRITELN('************** ProgramOne **************');
    TwoLines;
    WRITELN('************** ProgramOne **************');

END.
```

The example program imports ModuleTwo, which declared the procedure named TwoLines. Here is the source of ModuleTwo, which was compiled and stored in the library (object-code) file named MOD_2.CODE.

Source Listing of MOD_2.CODE

```
MODULE ModuleTwo;

IMPORT ModuleThree;

EXPORT
    PROCEDURE TwoLines;

IMPLEMENT

    PROCEDURE TwoLines;
        BEGIN
            WRITELN('I came from ModuleTwo and brought this:');
            ThirdLine;
        END;

END.
```

ModuleTwo exports procedure TwoLines, which is used by ProgramOne. It also imports ModuleThree, which declares procedure ThirdLine and is in the library (object-code) file named MOD_3.CODE.

Source Listing of MOD_3.CODE

```
MODULE ModuleThree;

EXPORT
    PROCEDURE ThirdLine;

IMPLEMENT

    PROCEDURE ThirdLine;
        BEGIN
            WRITELN('I came from ModuleThree');
        END;

END.
```

This module exports procedure ThirdLine, which is imported by ModuleTwo. Notice that it does not import any modules.

Here are the results of running the program.

```
************** ProgramOne **************
I came from ModuleTwo and brought this:
I came from ModuleThree
************** ProgramOne **************
```

Here is what happens when you run ProgramOne. First, ProgramOne prints two blank lines and then the line of asterisks that contains its name. The procedure TwoLines, imported from ModuleTwo, is then called; it prints the message: `I came from ModuleTwo and brought this`:. Procedure ThirdLine, imported from ModuleThree, is then called; it prints the message: `I came from ModuleThree`. Control is then returned to TwoLines and then to the program, which again prints out its name in asterisks.

Let's take a look at what is needed in order for you to compile and run the program.

Compiling and Running the Example Program

When a program (or module) imports modules, the imported modules must be accessible at two times:

- When the program is compiled.

- When the program is loaded and run.

Let's take a look at what happens at these two times.

How the Compiler Finds Imported Modules

At compile time, the Compiler searches for each module imported by the source program (or module); more specifically, it searches to find each module's "interface text." Here is the order of the places where the Compiler looks in search of interface text:

1. In the source text being compiled. (The source text of modules and programs can be combined into one source file, as long as the modules precede the program and are in proper sequence.)

2. In object files specified in a SEARCH Compiler option.

3. In the object file currently designated as the System Library.

(A module's interface text consists of the MODULE name, the IMPORT section, if present, and EXPORT section; these sections are part of the object module produced when the module was compiled or assembled. See the subsequent section called Getting Detailed Object File Information and the Compiler or Assembler chapters for a more complete description of interface text.)

Here is a strategy (and the method actually used) for compiling these source modules and program. (Note that you will be learning these Librarian operations in the subsequent examples given in this chapter, so you will probably want to perform this compilation exercise *after* working through the examples using the object modules and program).

1. Compile ModuleThree first (MOD_3.TEXT); call the resulting object file MOD_3.CODE for simplicity. Since this module does not import any others, it will be compiled with no need to search for any imported module's interface text.

2. Use the Librarian to add the resultant object module (MOD_3.CODE) to the library file currently designated as the System Library. (Actually, you will be creating a new library into which you will place the modules in the current System Library and ModuleThree; this type of operation is subsequently explained in this chapter.)

3. After merging these two libraries (into a third new library), you will need to do one of two things: use the What command to make the resultant library the System Library; or use the Filer to change the resultant library's name back to the name of the current System Library.

4. Next, compile ModuleTwo (MOD_2.TEXT); call the resulting object file MOD_2.CODE. The external references to ModuleThree will be resolved when the Compiler finds the object ModuleThree in the System Library.

5. Then place this compiled module in the System Library as in steps 2 and 3.

6. Compile the program (PROG_1.TEXT). Since both object modules upon which this program depends are in the System Library, they will be accessed automatically by the Compiler when the program is compiled.

7. Run the program. The loader automatically looks in the System Library in order to resolve the external references; it loads the modules required to complete the program (in this case, ModuleTwo and ModuleThree).

Since the program and modules have already been compiled and the object files placed on the DOC: disc, we will not discuss other alternatives of making the source files accessible to the Compiler. (However, you are again encouraged to do this after learning how to use the Librarian.)

Let's look now at how the loader finds imported object modules when the program is to be loaded for execution.

How the Loader Finds Imported Modules

Since a compiled program contains no record of where the Compiler found the imported modules, the loader must find the imported object modules at load time. Here is the order of the places where the loader looks:

1. Modules that are part of the object file being loaded.

2. In modules already P-loaded in memory, which includes all INITLIB and Operating System modules. (The loader searches for these modules in reverse order to which they were P-loaded; in other words, the most-recently loaded modules are searched first.)

3. In the current System Library file.

In order to make all imported modules part of the object file that uses them (alternative 1 above), you have two choices:

- Combine the source modules into one *source* file (and compile it). You can use the Editor to add each imported module's source file to the source program. You can also use an INCLUDE Compiler option in the source program to include each imported module's source file in the compilation of the program.

- Combine the object modules into one *object* file. Use the Librarian to combine the program and imported modules into one object file; you can optionally Link the modules to save space.

With both of these methods, only the file containing the program need be loaded; and when the program is finished, the memory used by the modules can be reclaimed for other purposes. With P-loaded modules, this is not possible (without re-booting).

If you want to P-load modules to make them accessible to the loader, you will only need to P-load all modules which are not in one of the three places stated above. In the example modules already given, ProgramOne imports ModuleTwo, and ModuleTwo imports ModuleThree. In the second example that follows, you will be creating a library that contains these two modules and then P-loading the library. (You can alternatively P-load MOD_3.CODE and MOD_2.CODE, in that order, which does not require use of the Librarian.) The loader will then be able to link the modules contained in the library to any program that imports them at execution time.

In general, the most convenient way to use modules is to place them in the file that is currently designated as the "System Library," which is the third alternative shown above. (The default System Library is the file named "LIBRARY" found on the system volume at power-up. You can also change it with the What command and the Main Command Level.) This is probably the most common reason for using the Librarian. In the first example that follows, you will add modules ModuleTwo and ModuleThree to the LIBRARY file and then run the program.

Subsequent tutorials also describe unassembling these library files and creating system Boot files.

Entering the Librarian

The Librarian is provided on the ACCESS: disc shipped with the system. To use the Librarian, you will first need to put it on-line: either place the disc labeled ACCESS: in a drive, or copy the LIBRARIAN file to another location (such as a hard disc) and use the What command (at the Main Command Level) to specify this copy as the system Librarian. After doing either of these, pressing ⌐L⌐ directs the system to load and execute the Librarian program.

Here is the Librarian's main prompt:

```
Librarian   [Rev.  3.2  15-Jan-87]        15-Jan-87   8:11:58


Q  Quit
P  Printout  OFF  PRINTER:LINK.ASC
O       Output file:  (none)
B  write to Boot disk
H  file Header maximum size:          38




I        Input  file:  (none)




Copyright 1987 Hewlett-Packard Company.
command?
```

The commands shown on the left-hand side of the screen are invoked by pressing the corresponding key. You will see how to use all of them in the following tutorial discussions. All commands are summarized in the Librarian Command Reference.

Setting Up Mass Storage

You will often need two on-line mass storage volumes when using the Librarian. If you only have one volume in your system, you may need to set up a memory volume. This discussion tells why two volumes may be needed and then outlines how to estimate the size of the volumes required.

When you combine the object modules in two libraries using the Librarian, you actually create a third (new) library and then copy into it the desired modules from the other two libraries. For instance, suppose that you want to add the CONFIG:RS232 module to the BOOT:INITLIB library file. You will first create a new library, and then add the existing INITLIB modules and the RS232 module to this new library. This new library must *not* be taken off-line during the entire process.

Thus, two separate volumes are often necessary for these two reasons:

- The destination module must *not* be taken off-line during this entire operation.

- The sum of *all* source libraries plus the new destination library may exceed the capacity of one volume.

Continuing with the preceding example, suppose that you have only one single-sided mini-disc drive on-line (the capacity is approximately 1050 sectors). The operation cannot usually be completed, because one mini disc is not large enough to contain the modules in the INITLIB file (let's assume 750 sectors), the RS232 module (approximately 25 sectors), and the new INITLIB file (*roughly* the sum of 750 and 25 sectors). You will need two volumes for the process.

If you don't have two disc drives (or one with sufficient space), you can create a memory volume. It is usually more convenient to use the memory volume as the destination volume. In this case, you could create one with a specified size of 400 blocks, or 200 Kbytes. (Remember that memory volume blocks are *512* bytes each, while mini-disc sectors are 256 bytes each.) See the Memvol command in the Overview chapter for more specific details on creating memory volumes.

The following examples assume that either you have two disc volumes on-line or that you have created a memory volume of sufficient size. For these examples, a memory volume of 100 blocks is sufficient.

Creating Libraries of Object Modules

To create libraries, you can combine either modules provided by HP or your own modules, or any combination of the two. Let's first look at adding modules to the System Library file.

Adding Modules to the System Library

A common way to use library modules is to add them to the current System Library file. Let's assume that it is the file named LIBRARY for present purposes, although you can change it to any file by using the What command at the Main Command Level. The procedure used to add modules to LIBRARY is very similar to that of storing modules in a user library, which is the next example.

Here is a brief summary of the steps required: first, make a new library file, and copy into it all of the modules currently in LIBRARY; next, add ModuleThree and ModuleTwo to the new file (in this case the order of modules is arbitrary, since the loader will load them in the right order); then replace the LIBRARY file with this new library; execute the program, and the modules are loaded automatically for you. The actual procedure is given below.

1. 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.) Now use the Librarian to create the new library.

2. Put the SYSVOL: disc (or the one containing the LIBRARY file) in the #3 drive. Press ⎡I⎤ and then type **#3:LIBRARY.** and press ⎡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 RND if you have not yet modified the LIBRARY file.) If you have a printer, you can press ⎡F⎤ to list all of the modules in the Input library.

3. (For this example, we will assume that you are using unit #4 as the second volume; however, if the LIBRARY file is small enough, you can also put the new library file on drive #3. We will also assume that the destination volume has enough room for the new library file.)

   Press ⎡O⎤ and enter **#4:NEWLIB.** as the Output file. Again, a trailing period prevents the .**CODE** suffix from being appended to the file name. If you are using a memory volume, use the unit number of the memory volume.

   (If you are using a disc, this disc must *not* be removed until you have finished creating the new NEWLIB file.)

4. Press ⎡E⎤ to enter the Edit mode. You should now see this prompt (in the middle of the screen):

   ```
   F  First module: RND
   U  Until module: (end of file)
   ```

5. You can now transfer all modules in the Input file to the Output file, including the last module, by pressing ⎡C⎤ (for Copy).

6. When the preceding transfer is complete, press ⬚A to append a module to the NEWLIB Output file. The Librarian prompts with **Input file:**. Put the DOC: disc, or whichever disc now contains ModuleThree, in Unit #3 (*not* #4, which must **not** be removed). Enter **#3:MOD_3** as the Input file.

7. The Librarian now prompts with **Enter list of modules or = for all**. Enter = for all. After ModuleThree has been transferred to the NEWLIB library, the Librarian prompts with **Append done, <space> to continue**. Press the spacebar to clear the prompt.

   Now use steps 6 and 7 again to copy ModuleTwo (in file MOD_2.CODE) into the NEWLIB file.

8. Now that all modules have been added to the NEWLIB file, press ⬚S to stop editing and ⬚K to keep the file.

9. You should now verify that the modules were indeed copied to the Output file. Press ⬚I and enter **#4:NEWLIB.** 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.

10. If all modules are present, then press ⬚Q to Quit the Librarian.

11. Now you have one of two options to make this library the System Library: you can use the What command at the Main Level to specify the file named NEWLIB (on the destination volume) to be the System Library; or you can replace the LIBRARY file on the SYSVOL: disc with this file. If you choose the second option, it is probably better to keep the current copy of LIBRARY on the disc; you should first use the Filer to Change its name to something like OLDLIB and then Filecopy the NEWLIB file onto the SYSVOL: disc, changing its name to LIBRARY.

12. Make sure that the System Library file is on-line, and then eXecute or Run the program.

As the program is loaded, the imported modules will also be loaded automatically. Here are the results of running the program.

```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

After the program has completed execution, the memory used by both program and modules can be used for other purposes.

As you can see, the System Library is a special library of object modules that is automatically accessed by the linking loader at program execution time (and by the Compiler at compile time). Because of this automatic access, you do not need to use the Permanent-load command to access this library's contents. This library would normally store those modules often used in your programs. Further descriptions of using HP-supplied libraries are given in the *Pascal 3.2 Procedure Library* and *Pascal 3.2 Graphics Techniques* manuals.

Making Your Own Library

Since we created a library that contains the modules named ModuleTwo and ModuleThree in the preceding example, you already know what is required to make your own library. The only difference is that you will not be adding the current LIBRARY modules to your library.

Here is a brief summary of the steps you will take in this example: first, create a new library with the Librarian and add the example modules ModuleTwo and ModuleThree to it (as with the last example, the order of modules is arbitrary; since they are in one file, the loader will take care of loading them in the proper order); P-load this library; and execute or run the program. A more detailed procedure follows.

Note

During the transfer process, you must **not** move the destination disc (the one that contains the Output file).

---

1. From the Main Command Level, press ⎡L⎤ to enter the Librarian. Your screen should now display the Main Prompt for the Librarian.

2. Put the destination disc in drive #4. Then press ⎡O⎤, and type #4:USERLIB and press ⎡Return⎤ or ⎡Enter⎤ to enter the Output file specification.

3. Place the DOC: Disc into the #3: disc drive. Then press ⎡I⎤, and enter #3:MOD_3 as the Input file specification. You will see MOD_3.CODE displayed as the Input file. The first object module found in the object file, MODULETHREE, is also displayed. The computer is in Copying mode as shown by the word COPYING on the prompt.

4. Transfer the object module MODULETHREE using the T command. Since MODULETHREE is the only module in that file, the A command would have done the same job.

5. Repeat steps 3 and 4 to name MOD_2 as the Input file and Transfer the object module MODULETWO into your new library.

6. If you had other modules to transfer, you would repeat steps 3 and 4 as needed.

7. Press ⎡K⎤ to Keep the new file on the destination volume.

8. Press ⎡Q⎤ to Quit the Librarian and return to the Main Command Level.

9. If you P-loaded these modules as you worked through the preceding example, then you need to re-boot in order to fully test your new library (to ensure that the modules P-loaded in the preceding example aren't accessed instead).

10. Press ⎡P⎤ for the Permanent-load command. You are prompted: **Load what code file ?** Enter #4:USERLIB (you don't need a period if you didn't include one when you specified this file as the Output file).

11. Now press ⎡X⎤ to eXecute ProgramOne. Answer the **Execute what file?** prompt by entering #3:PROG_1 as the file specification.

The results of the executed program are shown below.

```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

As mentioned earlier, you could also have separately P-loaded ModuleThree and ModuleTwo, in that order, and then run the program. Or, as with the preceding example, you could also have added these modules to the System Library. You could also have used What at the Main Command Level to specify this library as the System Library. The method you use depends on factors such as these: whether you are developing and testing the modules; whether you are also using other modules in the System Library; who will be using the modules; and so forth.

Linking Object Files Together

The Librarian permanently links modules together by combining their module directories into a single directory. To see this process in action, you will be linking the two example modules and the example program together.

1. Put the ACCESS: disc in a drive and press ⌐L⌐ to run the Librarian.

2. Put the DOC: disc in the #3 drive. Press ⌐O⌐, and then type **#3:TEST_1** and press ⌐Return⌐ or ⌐Enter⌐ to enter the Output file specification. #3:TEST_1.CODE will be our linked library's name; **#3:TEST1.CODE** is now displayed as the Output file. When an Output file is named, the menu replaces the B and H command prompts with the **L Link** prompt. The Librarian also enters the COPYING mode.

3. Enter the LINKING mode by pressing ⌐L⌐. This is the first step in the two-step linking process. Your screen now displays a new command prompt, as shown below.

```
Librarian  [Rev.  3.2    15-Jan-87]           21-Jan-87  8:05:08

Q  Quit
P  Printout  OFF  PRINTER:LINK.ASC
O       Output file:   TEST_1.CODE
C  Copy                              LINKING
N  Name of new module:   (none)
R  Relocation base:          0
G  Global base:              0
S  Space for patches:
D  output Def table?  YES
X  copyright notice:

I       Input file:   (none)




command?
```

4. Press ⌐N⌐ and enter **NEWNAME** as the new module name. If you did not do this, the object module contained in the new object file, TEST_1.CODE, would be the module name of the first module transferred. To avoid confusion, use "NEWNAME".

5. Press ⌐I⌐ and enter **#3:PROG_1** as the Input file. (The ".CODE" suffix is automatically appended to the Input file's name.) PROG_1.CODE is now the Input file.

6. Press ⌐A⌐ to transfer all object modules contained in PROG_1.CODE into TEST_1.CODE. Since PROG_1.CODE contains only one module, ⌐T⌐ would have done the same job.

7. Repeat steps 5 and 6 to transfer MOD_2.CODE and MOD_3.CODE. When all files are transferred, final linking must be done.

8. Press ⌷L⌷ to complete the linking process. This is the second step in the linking process. Remember that **all** object modules must be on-line when you complete the linking process.

9. Press ⌷K⌷ to Keep the new file, and press ⌷Q⌷ to Quit the Librarian and return to the Main Command Level.

To see that everything works, execute your new program. From the Main Command Prompt press ⌷X⌷, and then enter **TEST_1** as the file specification. The ".CODE" is automatically added to the file name. Your screen should now display the the following:

```
*************** ProgramOne ***************
I came from ModuleTwo and brought this:
I came from ModuleThree
*************** ProgramOne ***************
```

The benefit gained over merely combining modules into one library is that linking modules together reduces the amount of space required to store the library.

Subtle Points about Linking

There are several subtle side effects that occur when modules are linked that should be discussed here.

- When you link object modules, the interface text is removed. Thus, linked modules **cannot** be searched by the Compiler when it is attempting to satisfy IMPORT statements; however, these modules can be used by other modules at load time by P-loading them or placing them in the System Library. (Remember that you can also keep a copy of the unlinked object module which can, of course, be imported by other modules at compile time.)

- The linking process always produces relocatable object code. This code has been relocated to the values specified by Global base and Relocatable base, but it will be relocated again when it is loaded for execution. For this reason, you don't need to specify Global base and Relocatable base — just leave them zero.

- If two or more programs are linked together into one object file, the resultant file contains code with only *one* start address (rather than the two that you began with). Contrast this to the situation in which you put two programs in an object file; when this file is executed, the two programs get executed separately in the order encountered in the file. This is the reason that you cannot link the INITLIB modules together; it is actually a set of programs and modules in a library file.

- After linking, most programs will still have unsatisfied external references (such as calls to the File System read and write routines). These unsatisfied references do not cause error messages; they are satisfied by the linking loader as the program is prepared for execution. These system routines are not part of the compiled or linked program; rather, the entire operating system looks to the linking loader like a group of P-loaded user libraries.

- **NOTE**: do not create any module whose DEF table is bigger than 65 534 bytes. The Librarian can create it, but after it's created neither the Librarian nor the linking loader can work with that module. This can only happen if you have an extremely large program that is linked together while keeping the DEF's. You can tell how big the DEF table is by looking at the header generated by any Unassemble command in the Librarian. A work-around is to break that one module into more than one module, probably in the same file.

Summary of Linking Object Files

Note

All input modules must remain on-line for the duration of steps 6 through 9. The output file must be on-line for steps 3 thru 10.

---

1. Enter the Librarian.

2. Be sure the disc containing the file to be linked is in the appropriate disc drive.

3. Specify an Output file name.

4. Press ⌊L⌉ to begin the linking process

5. Name the new module with the Name Command.

6. Specify the Input file containing the modules you want to link.

7. Transfer only those modules you want into the new Output file using the All and Transfer commands.

8. Repeat steps 6 and 7 until all modules are transferred.

9. Press ⌊L⌉ to complete the Linking process.

10. Press ⌊K⌉ to keep your output file.

11. Press ⌊Q⌉ to quit the Librarian

Getting Detailed Object File Information

Let's unassemble the file MOD_2.CODE to see how the Librarian provides detailed information about a code file. It is best to have a printer on-line while unassembling; however, if you don't, you can declare a Printout file as described in the following procedure.

1. The Librarian is on the ACCESS: disc shipped with your system. To access the Librarian, you will need to put this disc on-line, or copy the file to a disc that is on-line, or P-load the file. After the file is on-line, press $\boxed{L}$ (at the Main Command Level) to load the Librarian subsystem into the computer.

2. If you don't have a printer on-line, then you must specify a file to which the unassembled information is sent. Press $\boxed{P}$ (for Printout) and enter a file specification; if no suffix or trailing period is included, then ".TEXT" is appended to the file name. The screen will be updated to show that the printout device is ON and that it is the file you specified.

3. Press $\boxed{I}$ and enter #3:MOD_2 as the Input file. No Output file is needed.

4. Press $\boxed{U}$ to get into the Unassemble mode.

Your screen should now show the Librarian Unassemble menu.

```
Librarian  [Rev.  3.2   15-Jan-87]          21-Jan-87  9:45:02

Q  Quit
S  Stop unassembling
T  print import Text
E  print Ext table
D  print Def table
A  unassemble all (Assembler  conventions)
C  unassemble all (Compiler   conventions)
P  PC    range     (Assembler  conventions)
L  Line range      (Compiler   conventions)




unassemble option?


```

When the first command key is pressed, an information header is printed along with the desired information. This header is printed only this one time. An internal counter keeps track of the line count and prints a page heading at the top of every new page. If you change the placement of the printer paper, you may waste some paper when the counter sends a form-feed to the printer. When you quit, a final form-feed is sent to the printer automatically.

The Text and Table Commands

Use these commands to obtain Interface Text and REF and DEF tables of modules.

The Print Import (or Interface) Text Command

Pressing ⊤ prints the interface text (DEFINE SOURCE) of the module, if any. In a compiled module, the DEFINE SOURCE portion consists of the text in the MODULE, IMPORT (if present), and EXPORT declarations; in an assembled module, this text consists of the lines containing the SRC pseudo op. (Note that any comments and indentation have been removed.)

```
Librarian   [Rev.  3.2  15-Jan-87]        23-Jan-87  7: 6:51     page 1


MODULE    MODULETWO    Created 23-Apr-84
NOTICE:  (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Directory size    174 bytes
  Module size      3072 bytes
  Module NOT executable
  Code base              0     Size          104 bytes
  Global base            0     Size            0 bytes
  EXT    block   5    Size          72 bytes
  DEF    block   3    Size          90 bytes
  EXPORT block   1    Size          74 bytes
  There are          1 TEXT records


    DEFINE SOURCE of 'MODULETWO':

MODULE MODULETWO;
IMPORT ModuleThree;

EXPORT
PROCEDURE TwoLines;

END;
```

The Print EXT Table Command

Pressing ⎡E⎤ prints the table of External symbols the module references. Detailed information on the EXT table may be found later in this chapter.

```
    EXT table of 'MODULETWO':

      FS_FWRITELN
      FS_FWRITEPAOC
      MODULETHREE_THIRDLINE
      SYSGLOBALS
```

The Print DEF Table Command

Pressing `D` prints the table of symbols the module itself defines. Detailed information on the DEF table may be found later in this chapter.

```
DEF table of 'MODULETWO':

    MODULETWO                       Gbase
    MODULETWO_MODULETWO             Rbase+102
    MODULETWO_TWOLINES              Rbase+2
    MODULETWO__BASE                 Rbase
```

The Unassemble Commands

There are two conventions used when unassembling object files: Compiler and Assembler. The reason for this is that the Compiler and Assembler use different conventions for the object code that they generate.

The Compiler generates code so that each procedure begins with a `TRAP #1` or a `LINK #n,A6` and ends with a `JMP` or `RTS`. The Librarian uses this information to assume that everything from the beginning of the file to the first `TRAP #1` or `LINK` is a constant. From the end of the procedure to the next `TRAP #1` or `LINK` is also unassembled as constants. Everything else is unassembled as instructions. The Assembler convention assumes that everything is an instruction.

Note

All Unassemble commands require a printer unless a destination file is specified with the P command.

The Unassemble All (Compiler convention) Command

Pressing `C` directs the Librarian to unassemble the specified object module using the Compiler convention described above. You can use this command on files that were created by either the Assembler or Compiler. Here are the results of using this command with the MOD_2.CODE compiled object file.

```
Librarian  [Rev.  3.2  15-Jan-87]       23-Jan-87   9:58:42     page 1

MODULE     MODULETWO     Created 23-Apr-84
NOTICE:    (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Directory size     174 bytes
  Module size       3072 bytes
  Module NOT executable
  Code base                0      Size         104 bytes
  Global base              0      Size           0 bytes
  EXT    block   5     Size          72 bytes
  DEF    block   3     Size          90 bytes
  EXPORT block   1     Size          74 bytes
  There are            1 TEXT records
```

```
TEXT RECORD #              1  of 'MODULETWO':
  TEXT start block    2        Size           104 bytes
  REF  start block    4        Size            24 bytes
  LOAD address        Rbase


       0  0000               dc.w 0          or dc.b 0,0        or dc.b '  '
- - - - - - - - - - - - - - - - - - - - - - - - - - MODULETWO_TWOLINES
       2  4E41 0000          trap #1,#0
       6  2F2D FFA6          move.l SYSGLOBALS-90(a5),-(sp)
      10  2F17               move.l (sp),-(sp)
      12  487A 0030          pea Rbase+62
      16  3F3C 0027          move.w #39,-(sp)
      20  3F3C FFFF          move.w #-1,-(sp)
      24  4EB9 0000          jsr FS_FWRITEPAOC
          0000
      30  4AAD FFEA          tst.l SYSGLOBALS-22(a5)
      34  6702               beq.s Rbase+38
      36  4E43               trap #3
      38  4EB9 0000          jsr FS_FWRITELN
          0000
      44  4AAD FFEA          tst.l SYSGLOBALS-22(a5)
      48  6702               beq.s Rbase+52
      50  4E43               trap #3
      52  4EB9 0000          jsr MODULETHREE_THIRDLINE
          0000
      58  4E5E               unlk a6
      60  4E75               rts
      62  4920               dc.w 18720      or dc.b 73,32      or dc.b 'I '
      64  6361               dc.w 25441      or dc.b 99,97      or dc.b 'ca'
      66  6D65               dc.w 28005      or dc.b 109,101    or dc.b 'me'
      68  2066               dc.w 8294       or dc.b 32,102     or dc.b ' f'
      70  726F               dc.w 29295      or dc.b 114,111    or dc.b 'ro'
      72  6D20               dc.w 27936      or dc.b 109,32     or dc.b 'm '
      74  4D6F               dc.w 19823      or dc.b 77,111     or dc.b 'Mo'
      76  6475               dc.w 25717      or dc.b 100,117    or dc.b 'du'
      78  6C65               dc.w 27749      or dc.b 108,101    or dc.b 'le'
      80  5477               dc.w 21623      or dc.b 84,119     or dc.b 'Tw'
      82  6F20               dc.w 28448      or dc.b 111,32     or dc.b 'o '
      84  616E               dc.w 24942      or dc.b 97,110     or dc.b 'an'
      86  6420               dc.w 25632      or dc.b 100,32     or dc.b 'd '
      88  6272               dc.w 25202      or dc.b 98,114     or dc.b 'br'
      90  6F75               dc.w 28533      or dc.b 111,117    or dc.b 'ou'
      92  6768               dc.w 26472      or dc.b 103,104    or dc.b 'gh'
      94  7420               dc.w 29728      or dc.b 116,32     or dc.b 't '
      96  7468               dc.w 29800      or dc.b 116,104    or dc.b 'th'
      98  6973               dc.w 26995      or dc.b 105,115    or dc.b 'is'
     100  3A00               dc.w 14848      or dc.b 58,0       or dc.b ': '
     102  4E75               dc.w 20085      or dc.b 78,117     or dc.b 'Nu'
```

The Unassemble All (Assembler Convention) Command

Pressing [A] will cause your computer to unassemble the specified object module using the Assembler convention described above. You can use this command on files that were created by either the Assembler or Compiler.

Note

Use of the Assembler convention may produce unpredictable results, because under this convention there is no way to tell code from data. Files produced by the Compiler and unassembled under the Compiler convention will almost always produce reasonable results.

Here is the unassembly of the MOD_2.CODE object file using the Assembler convention. Notice that, with Assembler convention, the first two bytes ($0000) are assumed to be code; with Compiler convention they are assumed to be data (remember that the Compiler convention assumes that anything until the first **TRAP #1** or **LINK #n,A6** is assumed to be data). Notice also that the module heading shows that this object module was produced by the Compiler.

```
Librarian [Rev.  3.2  15-Jan-87]       23-Jan-87 10: 1:34 page 1

MODULE     MODULETWO    Created 23-Apr-84
NOTICE:   (none)
  Produced by Pascal Compiler of 23-Apr-84
  Revision number 3
  Directory size    174 bytes
  Module size      3072 bytes
  Module NOT executable
  Code base                0      Size         104 bytes
  Global base              0      Size           0 bytes
  EXT    block  5   Size          72 bytes
  DEF    block  3   Size          90 bytes
  EXPORT block  1   Size          74 bytes
  There are           1 TEXT records


TEXT RECORD #          1  of 'MODULETWO':
  TEXT start block   2       Size         104 bytes
  REF  start block   4       Size          24 bytes
  LOAD address       Rbase
```

```
0    0000 4E41      ori.b #65,d0
4    0000 2F2D      ori.b #45,d0
8    FFA6           dc.w -90        or dc.b 255,166     or dc.b ' '
10   2F17           move.l (sp),-(sp)
12   487A 0030      pea Rbase+62
16   3F3C 0027      move.w #39,-(sp)
20   3F3C FFFF      move.w #-1,-(sp)
24   4EB9 0000      jsr FS_FWRITEPAOC
     0000
30   4AAD FFEA      tst.l SYSGLOBALS-22(a5)
34   6702           beq.s Rbase+38
36   4E43           trap #3
38   4EB9 0000      jsr FS_FWRITELN
     0000
44   4AAD FFEA      tst.l SYSGLOBALS-22(a5)
48   6702           beq.s Rbase+52
50   4E43           trap #3
52   4EB9 0000      jsr MODULETHREE_THIRDLINE
     0000
58   4E5E           unlk a6
60   4E75           rts
62   4920           lea -(a0),a4
64   6361           bls.s Rbase+163
66   6D65           blt.s Rbase+169
68   2066           movea.l -(a6),a0
70   726F           moveq #111,d1
72   6D20           blt.s Rbase+106
74   4D6F 6475      lea 25717(sp),a6
78   6C65           bge.s Rbase+181
80   5477 6F20      addq.w #2,32(sp,d6.l)
84   616E           bsr.s Rbase+196
86   6420           bcc.s Rbase+120
88   6272           bhi.s Rbase+204
90   6F75           ble.s Rbase+209
92   6768           beq.s Rbase+198
94   7420           moveq #32,d2
96   7468           moveq #104,d2
98   6973           bvs.s Rbase+215
100  3A00           move.w d0,d5
102  4E75           rts
```

The Line Range (Compiler Convention) Command

Pressing ⌷L⌷ causes two prompts to be displayed. The computer needs to know the line number range to unassemble. The code will then be unassembled up to, but not including, the upper range value. The object module must have been compiled using the $DEBUG ON$ Compiler option to be unassembled with this command.

The PC Range (Assembler Convention) Command

Pressing ⌷P⌷ causes two prompts to be displayed. The computer needs the location counter values of the segment of code you want unassembled (relative to the relocation base of the module). The code will then be unassembled up to but not including the upper location counter value.

Creating a New Boot File

At power-up, the Boot ROM searches mass storage for system Boot files: Boot ROMs 3.0 and later versions search all mass storage devices on-line and let you choose which Boot file you want; earlier Boot ROMs choose the first Boot file found on the right-hand internal disc drive. A Boot file is then loaded by the Boot ROM. The Boot file in turn may load other parts of a system. (For further details of how this system boots, see the discussion called The Booting Process in the Special Configurations chapter.)

The ⟦B⟧ command is used to create Boot files. This is an advanced option and should only be used if you have a clear understanding of system generation.

The following is an overview of the system generation process using the ⟦B⟧ command.

Note

The B command cannot create boot files in WS1.0 directories.

---

1. Use the Editor to produce the programs and modules that will make up the new boot program. Both Assembler language and Pascal modules may be used.

2. Assemble the Assembler language modules and compile the Pascal modules.

3. Use the Librarian to Link the code files together as desired. Be sure to specify the global and relocation bases. In addition, this file must have no external references. Note that the first program linked will provide the start address for the Linked file. This start address will also become the start execution address of the system Boot file at boot time.

4. Keep the linked file.

5. Specify the linked file as the Librarian Input file.

6. Now press ⟦B⟧ to properly place the module name in the destination's directory and format the code for use by the boot ROM. The B command moves the cursor up to the Output file prompt.

7. Specify the Output file as SYSTEM_xxx (the xxx can be any characters syntactically allowed for file names. With Boot ROMs 3.0 and later, the name can be SYSxxxxxxx; see Re-Naming BOOT: Files in the Special Configurations chapter for examples). Be sure to append a trailing period to the file name to keep a suffix from automatically being appended to the name.

Note

To create a Boot file which will be recognized on an HFS disc, it is necessary to follow this process, then, using the OSINSTALL utility described in Chapter 21, convert the format to that recognized by HFS.

---

8. Transfer the Input file into the boot file. This copies the code file.

9. Press ⌐B⌐ again to finish the Boot operation.

10. You can now either Quit the Librarian or power up and test your new system.

Librarian Command Reference

The Librarian command set consists of single-letter commands allowed when the letter prompts are displayed on the screen. You press the corresponding key to cause the command to be executed.

⌐A⌐   In Copying and Linking modes, this command transfers All modules from the Input file to the Output file.

            In Edit mode, this command is used to Append modules to the Output file.

            In Unassemble mode, this command directs the Librarian to unassemble the Input file using Assembler conventions.

⌐B⌐   The Boot command is used to create code files that are loadable by the Boot ROM. The Boot command is given instead of the Output file command. The Input modules are then combined in a format that is bootable. This command should only be used by system designers. A boot file must be a self-contained processor environment. It must be stored on a LIF or SRM volume and be named SYSTEM_xxx, where xxx represents any combination of characters. If you have Boot ROM 3.0 or later version, the file can be named SYSxxxxxxx or stored on an SRM system under the /SYSTEMS directory. For systems using HFS, the OSINSTALL utility must be used to convert the LIF format Boot file into a suitable format for HFS discs.

⌐C⌐   In the Link mode, this command returns you to Copy mode. While in Copy mode, you can combine modules into a library without Linking. This mode can be used to add modules to (or remove them from) the System Library, your own library, or INITLIB (the Initialization Library file which is executed during the boot process).

            In Edit mode, this command Copies the First module up to (but not including) the Until module to the Output file.

            In Unassemble mode, this command unassembles the Input file according to Compiler conventions.

⌐D⌐   In Link mode, this command controls whether or not the DEF table is included in the Output file. If the Output file is to be Linked to another file later, the DEF table must be left in the Output file. If the Output file is not to be Linked, you can save memory space by removing the DEF table. A YES includes the DEF table, a NO removes it. Pressing ⌐D⌐ toggles between these two choices.

            In Unassemble mode, this command prints the DEF table.

⌐E⌐   This command is available when you have specified both Input and Output files. It puts you into Edit mode, which allows you to combine modules in the Input file with Append modules and place them into the Output file (while either Copying or Linking).

            In Unassemble mode, this command prints the Ext table.

F This command prints the File directory of the Input file on the current Printout file (external printer or file). It doesn't matter whether the **Printout** prompt is **ON** or **OFF**; the printout will be sent to the Printout file.

In Edit mode, this command is used to specify the First module to be transferred to the Output file. (The First module must precede the Until module in the Input file.)

G In Linking mode, designate the Global base address (most useful when preparing a file for use as a system Boot file).

H The Header command allows you to change the size of the library header. From 1 through 18 module entries require only one header block, so a header size of 18 is the minimum. If you specify less (but not 0), you will still be given 18. From 19 through 38 module entries requires two header blocks. This is the default header size. The specification is made in units of module entries — not blocks. The Librarian calculates how many blocks are necessary to maintain the number of modules you specify and then gives you the maximum number of entries that will fit in that number of blocks.

I Name the Input file containing the modules you want to transfer to the Output file. ".CODE" is automatically appended to the file name unless suppressed by a trailing period (or by the presence of another standard suffix). This prompt can be used many times to collect modules from several object files into your new object file.

K Keep the Output file. Close and lock it into the directory, purging any old file of the same name.

L Enter Linking mode or finish Linking.

In Unassemble mode, you can use this command to unassemble a section of code defined by two Line values using the Compiler convention. The code must have been compiled using the $DEBUG ON$ Compiler option.

M Enter the specific Module name you want to transfer. (The first module in the Input file is automatically displayed when an Input file is specified.)

N In Linking mode, name the New object module to be created by the Librarian. If you do not specify a new module name, the name of the first module transferred will be used.

O Name your Output file. ".CODE" is appended automatically unless suppressed by a trailing period (or standard suffix) in the file name. This Output file must remain on-line throughout the process of transferring modules to it.

P This command is used to turn the Printout option ON or OFF (the default is OFF) and to select a Printout file. Pressing **P** and **Return** or **Enter** turns the option ON. With the option ON, the device to which the information is sent is shown on the screen. The default Printout file specification is "PRINTER:LINK.ASC" unless you specify another. ".TEXT" is automatically appended to the file name unless it is suppressed by a trailing period (or standard suffix) in the file name.

Before any information is sent to the Printout file, the Librarian first sends heading information to the device. When Linking, you will get a map of all Linking done by the Librarian. This option does not affect any information sent to the Printout file by the Unassemble commands.

In Unassemble mode, this command allows you to unassemble (using Assembler conventions) a section of code defined by two location counter range values.

[Q] Quit the Librarian and return to the Main Command Level.

[R] In Linking mode, designate the Relocation base address to be used. Usually used only when creating a boot file.

[S] In Linking mode, this command assigns Space for patches. To save execution time and memory space, the Compiler can be made to use PC-Relative addressing instead of Long-Absolute addressing. This is done with the Compiler option $CALLABS OFF$. The PC-Relative addressing mode has an address range of − 32 768 through 32 767 bytes; if the referenced procedure is out of this range, an error will occur at load or link time. This error prints an error message naming the module having the link out of range. To fix this, relink the modules adding patch space between them as needed. The number of bytes needed depends on the particular module. As a rule of thumb, begin with a patch of 100 bytes.

In Edit or Unassemble modes, this command Stops the Edit or Unassemble session and returns to the Librarian's main prompt. (This will not stop an ongoing Unassemble; however, the [Stop] key will.)

[T] In Copying and Linking modes, this command Transfers the object module currently named in the Transfer prompt to the Output file.

In Unassemble mode, this command prints the interface Text (DEFINE SOURCE) of the current Input module.

[U] Enter the Unassemble mode.

In Edit mode, this command allows you to specify the Until module. If you enter a null response (by pressing [Return] or [Enter] with no file specification), then **(end of file)** is displayed; a subsequent Copy will copy all remaining modules in the Input file (i.e., up to the end of the file) to the Output file.

[V] This command gets you into the Verify mode. This mode displays the name of each module in the Input file and allows you to Transfer it to the Output file (press [T]), to Unassemble it, or to not transfer or unassemble it (press the space bar) and step to the next module name. After all module names have been displayed, you automatically leave this mode. To re-verify the file's contents, press [V] again.

[X] In Linking mode, this command allows you to enter a copyright notice as part of the Output file. The notice is part of the heading information sent to the Printout file. The notice can be up to 255 characters long.

Glossary of Object Code Terminology

Here are detailed definitions of the terms used in this manual regarding object-code library files.

DEF table (Definition Symbol Table)

There is only one DEF table per module. It contains one DEF record for each symbol which is exported from the module. The DEF table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The DEF table is contiguous over its length, which means that individual DEF records within the table may cross block boundaries.

Each DEF record has two parts. The first part is a packed string containing the name of the symbol which is defined. The string begins and ends on a word (even-byte) boundary. If the string length is odd, then an extra byte is added to the end for padding so that the next part of the DEF record will begin on a word boundary.

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. GVR is defined later in this section.

The value extension is 4 bytes or 8 bytes long, according to the data size field. The value of the symbol is defined to be the value extension plus what ever references are specified by the primary type and any Reference Pointers that may exist. The value extension must be present.

DEF record

low

LEN = 6	S
Y	M
B	O
L	padding
flags	len = 8
value (high part)	
value (low part)	
ref pointers (....)	

First part (LEN = 6 through L/padding)

Second part (flags/len = 8 onward)
(len is Second part length)

(GVR includes any number of reference pointers)

high
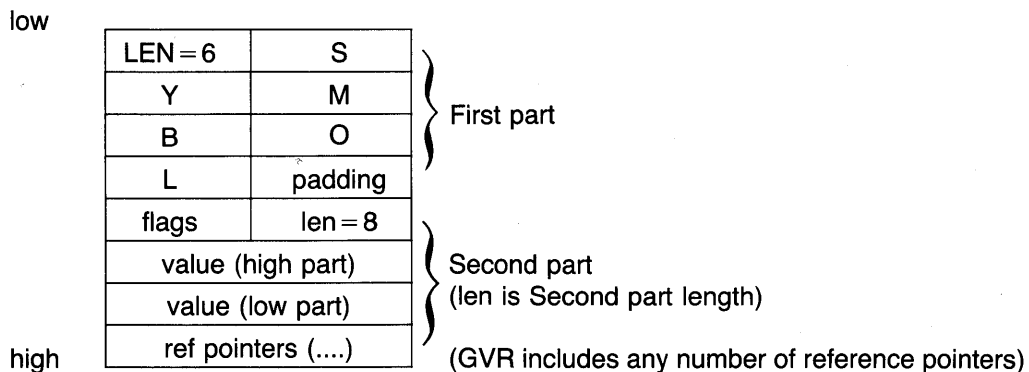
Caution: do not create any module whose DEF table is bigger than 65 534 bytes. The Librarian can create it, but after it's created neither the Librarian nor the linking loader can work with that module. This can only happen if you have an extremely large program that is linked together while keeping the DEF's. You can tell how big the DEF table is by looking at the header generated by any Unassemble command in the Librarian. A work-around is to break that one module into more than one module, probably in the same file.

DEFINE SOURCE

This is the section of an object module that is searched by the Compiler when the module is imported (also called "interface text"). With Pascal modules, the DEFINE SOURCE consists of the declarations made by the reserved words MODULE, IMPORT (if present), and EXPORT. With Assembler modules, it consists of the lines defined by the SRC pseudo op, which are intended to serve the same function as in Pascal modules (however, it may be any arbitrary text).

There may be one table of DEFINE SOURCE per module. It begins on a block boundary, which is given in the module directory. The length is also given in the directory.

EXT Table (External Symbol Table)

The EXT table contains records (Pascal strings), each of which is the name of a symbol referenced in this module, but not defined in it (i.e., these symbols are declared in another module which this one imports and to which this module is linked at load time).

There may be one EXT table per module. The EXT table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The EXT table is contiguous over its length, which means that individual EXT records within the table may cross block boundaries.

Each EXT record is a multiple of four bytes long. The first byte of each string is its length (according to the Pascal string type); thus strings may be from 1 to 255 bytes long. If strlen(string)+1 is not a multiple of 4, then 1 to 3 bytes are added as padding to make the EXT record extend to the proper boundary.

The first eight bytes of the EXT table are reserved. Thus, the first string in the table starts at an offset of 8 from the start of the table.

The EXT table is restricted to 65 532 bytes in length (plus the length of the last string). This is so that any entry in the table can be uniquely referenced by 14 bits; the reference is relative to the start of the table. See the description of the reference pointer.

EXT Record

left byte	right byte
LEN = 6	S
Y	M
B	O
L	padding

low (top), high (bottom)

This one is 8 bytes long.

The formula is:

EXTsize = len + 4 − (len mod 4)

EXPORT

EXPORT is a reserved word used in the Pascal Module. It is used to declare those procedures, functions, constants, types, and variables that are exported, or made available, to other modules that import the module.

Flags

Flags are used in the DEF table, in REF tables, and in the GVR. Their form is shown below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Primary Type		Data Size			Patchable	Value-Extend	Long Offset

primary type:	00 absolute	:no REFERENCE POINTERS follow
	01 relocatable	:no REFERENCE POINTERS follow
	10 global	:no REFERENCE POINTERS follow
	11 general	:one or more POINTERS follow

data size:	000 signed byte (8 bits)	−128..127
	001 signed word (16 bits)	−32768..32767
	010 signed long (32 bits)	−2147483648..2147483647
	011 (reserved)	
	100 unsigned byte (8 bits)	0..255
	101 unsigned word (16 bits)	0..65535
	110 (reserved)	
	111 (reserved)	

patchable: Indicates that the linker may patch a location in a TEXT record. Applicable only in a REF record and must be false everywhere else.

value extend:
0 No extension present, assume 0
1 Value extension present. Length is 4 bytes.
  Always true in DEF records.

long offset:
0 Use short form (1 byte) of offset field. Value is in the range 0..255 and specifies the total length of the GVR except in REF records.
1 Use long form (3 bytes) and offset field. Value is a 24 bit unsigned number in the range 0..16777215. Applicable only in some REF records.

Note

Data Size should be signed long everywhere except in a REF record.

General Value or Address Record (GVR)

The GVR is a variable length record which is intended to represent any absolute, relocatable, or linkable value.

```
TYPE DATATYPE = (sbyte, sword, sint, fltpt, ubyte,uword);
     RELOCTYPE = (absolute, relocatable, global, general);
GENERALVALUE = PACKED RECORD
   PRIMARYTYPE : RELOCTYPE;      (*allows quick indication
                                      of most common types*)
   DATASIZE : DATATYPE;          (*specifies 1,2 or 4 bytes, signed or not*)
   PATCHABLE,                    (*specifies self relative field in branch*)
   VALUEEXTENDED : BOOLEAN;      (*indicates valueextenion*)
   CASE LONGOFFSET : BOOLEAN OF  (*1 or 3 byte offset*)
      FALSE : (short:0..255);    (*unsigned 8 bits*)
      TRUE : long:0..16777215);  (*unsigned 24 bit value*)
END;
   VALUEEXTENTION = PACKED       (*present if value extended bit above is set*)
               RECORD
      CASE DATATYPE OF
          SBYTE,SWORD,SINT,
          UBYTE,UWORD :     (value integer);
END;
REFERENCEPTR = PACKED RECORD   (*one or more present if type = general*)
   ADDRESS : 0..16383;         (*multiply by 4 to get address of EXT symbol*)
   OP : (ADDIT,SUBIT);         (*add or subtract the modifying value*)
   LAST : BOOLEAN;             (*indicated end of list*)
END;
GVR = CONCATENATION            (*NOTE* This is pseudo pascal*)
   GENERALVALUE;               (*2 to 4 bytes of header info*)
   VALUEEXTENTION;             (*0 or 4 bytes of value*)
   ARRAY[zero or more] OF
       REFERENCEPTR;           (*list of EXT references*)
END;
```

IMPLEMENT

IMPLEMENT is a reserved word used in the Pascal Module. It is used as a flag to indicate the beginning of the module body. The module body can be either empty or may contain those constants, variables, procedures and functions used internally by the module. None of this information is available outside the module (unless it is also declared in the module's EXPORT section).

IMPORT

IMPORT is a reserved word used in the Pascal Module. It names the modules whose DEFINE SOURCE sections must be examined by the Compiler in order to resolve references to constants, variables, procedures, and functions exported by the modules. The Compiler uses a module's name in conjunction with names of constants, procedures, and functions declared in the module to generate EXT strings for which the loader will search (and link) at run time.

LIBRARIAN

The Librarian is a subsystem designed to manage HP Workstation Pascal and Assembler object files, link and unassemble object modules, and create system Boot files. It can merge object files containing object modules and optionally link the object modules together. It is the file named LIBRARIAN in your operating system, which can be changed with the Main Level's What command. It is accessed by pressing ⟪L⟫ from the Main Command Level.

Library

A library is an object file produced by the Assembler, Compiler, or Librarian. Its purpose is to contain object module(s).

LIBRARY

LIBRARY is a library file included with your operating system. During the boot process, this file (if on-line) generally becomes the System Library; you can also use the What command at the Main Level to specify any file as the System Library.

Only a few useful object modules are included in the file when you received it. Feel free to examine them with the Librarian. Other object modules are supplied with the system (in the IO and GRAPHICS libraries for example) and may be added to the LIBRARY.

Object File

An Object File is the unit of object code managed by the Librarian. It is made up of a Library directory and one or more object modules. The Assembler generates one object file from each source file assembled; the Compiler also generates one object file per invocation. 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

Each object module is made up of a module directory and a module body. The module body is made up of the following items:

One EXT table	A table of the symbols imported by the module.
One DEF table	A table of symbols exported by the module.
One DEFINE SOURCE Area	A software interface between the module and any program which imports it.
One or more Text-Record/REF-Table pairs	A text record consists of the constants and code instructions that make up the the program. The REF table is a directory of linkage information required by the TEXT record.

Pascal Module

HP Pascal allows source modules to be compiled separately into object modules. The object modules are generally not executable, but are used to complete other Pascal programs. Examples are given earlier in this chapter.

REF tables

Each REF table follows a TEXT record and is associated with that TEXT record. The REF table begins on a block boundary, which is specified in the directory for the module. Its length is also given in the directory. The REF table is contiguous over its length.

Each REF record is associated with one object (byte, word or integer) within the preceding TEXT record. There can be at most one REF record for a given object in the TEXT record. The REF records are ordered within the table according to the TEXT objects they reference.

The offset field specifies which text object is referenced. The first REF record gives an offset from the beginning of the TEXT record. Subsequent REF records give an offset from the object referenced by the previous REF record.

low	

flags	offset
offset (low part)	
Ref pointers	

high

Ref record is a GVR.

offset, 1 or 3 bytes, indicates next object in TEXT record.

Can include any number of Ref Pointers.

Reference Pointer

Bit 15 thru Bit 2	Bit 1	Bit 0
Address of an EXT Record Relative to Beginning of EXT Table	Add or Sub	End Flag
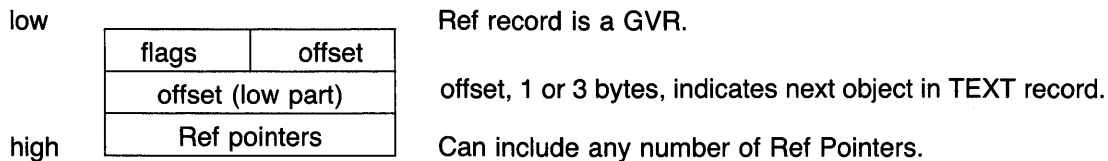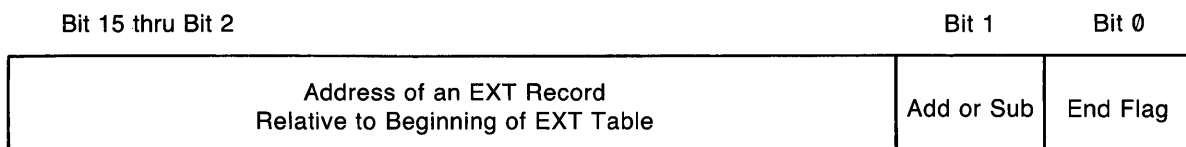
A REFERENCE POINTER is the relative address of an entry in the EXT table.

The add or sub flag indicates whether the value of the external symbol is to be added (0) or subtracted (1) from the GVR value in order to obtain the actual value. There may be any number of REFERENCE POINTERS in a GVR, and there may be more than one reference to the same EXT record. There may not, however, be both an add reference and a subtract reference to the same symbol, since these would cancel each other.

The end flag indicates whether there are any more REFERENCE POINTERS in the GVR. (0) indicates more to come, (1) indicates the end.

There are two special cases for the EXT address.

- Address 0 (bit pattern 00000000000000xx) refers to the relocation delta for the current module (i.e. new load address minus the old load address).

- Address 4 (bit pattern 00000000000001xx) refers to the global delta for the current module (i.e. new data address minus old data address)

Address 8 (bit pattern 00000000000010xx is the first valid reference to an external symbol.

There are REFERENCE POINTERS in a GVR only if the primary type field specifies general.

System Library

The System Library is a file that is automatically accessed by the Compiler at compile time and by the linking loader at execution time. Object modules stored in this object file are automatically available to any program importing them.

During the booting process, the LIBRARY file usually gets designated as the System Library; however, you can use the What command at the Main Level to specify any file. See LIBRARY above.

Text Record

A Text record is a contiguous section of code, beginning on a block boundary, which has an entry in the module directory. The length is also given in the directory. The text record can be any arbitrary data, but is usually the object code produced by the Compiler or Assembler.

The Debugger

9

Introduction

The Workstation Pascal System features a programming aid called the Debugger. As you probably have guessed, the major purpose of the Debugger is to make program debugging as painless as possible.

You may have already seen a reference to this Debugger when you got this message:

```
RESTART WITH DEBUGGER?
```

The question is in response to a user program generating *but not trapping* an "exception." You will learn how to answer the question in this chapter.

Here are some of the operations you can perform with the Debugger:

- Step through programs on a procedure, statement, or machine-instruction basis.

- Maintain a record of the statements which have already been executed (in order of execution).

- Examine any memory locations and CPU registers, and display the contents in any of the following formats: binary, octal, decimal, or hexadecimal integer; real number; alphanumeric character; and Assembler instruction.

- Set up "breakpoints" and "error traps" in the program, optionally displaying helpful information when each is encountered.

- Perform number-base conversions and integer arithmetic calculations.

The main emphasis of this chapter is to describe using the Debugger to debug Pascal programs. Debugging an Assembler-language program is more direct; that information is obtainable from the Debugger Reference Section.

Is the Debugger Loaded?

The Debugger is a very powerful subsystem, because it allows any user to access *everything* in the computer. It is therefore a potentially dangerous feature in the hands of users who don't know how to use it (or who you don't *want* to use it). For this reason, and for space considerations, it is *not* automatically loaded when you boot the system (as shipped). Therefore, you will need to load the Debugger before attempting to use it (or install it in INITLIB). (In most previous system versions, it was automatically loaded at boot time, as it was part of the INITLIB file). Loading the Debugger is explained in the following Sample Session section.

A Sample Session

This section describes methods for debugging Pascal programs with the aid of the sample program called DEBUG, supplied to you on the DOC (documentation) disc. The program is given in source-code and object-code form.

- DOC:DEBUG.TEXT — the source-code file

- DOC:DEBUG.CODE — the object-code file

The Example Program

A listing of the program is included here for reference. Note that a Pascal source program must contain the $DEBUG ON$ Compiler option if you want to have the ability to halt the program at particular line numbers. The effects of this option are further described in the Compiler chapter.

```
Pascal [Rev 3.2   1/15/87] DEBUG.TEXT              28-Jan-87 14:21:55 Page 1

     1:D          0 $DEBUG ON$ { Enable debugging. }
     2:S
     3:D          0 PROGRAM XYZ (OUTPUT);
     4:D          1    VAR
     5:D    -4    1       i         : INTEGER;
     6:D    -8    1       j         : INTEGER;
     7:D   -16    1       x         : REAL;
     8:D   -24    1       y         : REAL;
     9:D   -25    1       ch1       : CHAR;
    10:D   -26    1       ch2       : CHAR;
    11:S
    12:D          1    PROCEDURE Level_1;
    13:D          2       VAR
    14:D    -4    2          i       : INTEGER;
    15:D    -8    2          x       : INTEGER;
    16:D   -12    2          y       : INTEGER;
    17:S
    18:D          2       PROCEDURE Level_2b;
    19:C          3          BEGIN
    20*C          3             WRITE('Level 2b:   ');
    21*C          3             WRITE('  i=',i:2,'  x=',x:4);
    22*C          3             WRITELN('  ch1=',ch1:1);
    23*C          3          END;
    24:S
    25:D          2       PROCEDURE Level_2a;
    26:D          3          VAR
    27:D   -12    3             i, x, y  : INTEGER;
    28:S
    29:D          3          PROCEDURE Level_3;
    30:C          4             BEGIN
    31*C          4                IF i < 4 THEN
    32:C          5                   BEGIN
    33*C          5                      WRITE('Level 3:   ');
    34*C          5                      WRITE('  i=',i:2,'  x=',x:4);
    35*C          5                      WRITELN('  ch1=',ch1:1);
    36*C          5                      i:= i + 1;
    37*C          5                      Level_3;
    38:C          5                   END;
    39*C          4                IF ch1='a' THEN
```

```
40:C         5                    BEGIN
41*C         5                        ch1:= 'x';
42*C         5                        WRITE('Level 3:   ');
43*C         5                        WRITE(' i=',i:2,'  x=',x:4);
44*C         5                        WRITELN('  ch1=',ch1:1);
45:C         5                    END;
46*C         4                END;
47:S
48:C         3            BEGIN
49*C         3                WRITE('Level 2a:  ');
50*C         3                WRITE(' i=',i:2,'  x=',x:4);
51*C         3                WRITELN('  ch1=',ch1:1);
52*C         3                i:= 1; x:= 2; y:= 3;
53*C         3                Level_3;
54*C         3                i:= 4; x:= 5; y:= 6;
55*C         3                Level_2b;
56*C         3            END;
57:S
58:C         2        BEGIN
59*C         2            i:= 0; x:= 0; y:= 0;
60*C         2            WRITE('Level 1:    ');
61*C         2            WRITE(' i=',i:2,'  x=',x:4);
62*C         2            WRITELN('  ch1=',ch1:1);
63*C         2            Level_2b;
64*C         2            Level_2a;
65*C         2        END;
66:S
67:C       1 BEGIN
68*C       1     i:= 10;
69*C       1     x:= 20.0; y:= 30.0;
70*C       1     ch1:= 'a'; ch2:= 'b';
71*C       1     WRITE('Main:       ');
72*C       1     WRITE(' i=',i:2,'   x=',x:2:1);
73*C       1     WRITELN('  ch1=',ch1:1);
74*C       1     Level_1;
75*C       1     WRITE('Main:       ');
76*C       1     WRITE(' i=',i:2,'   x=',x:2:1);
77*C       1     WRITELN('  ch1=',ch1:1);
78*C       1 END.
79:S
```

No errors. No warnings.

Please Participate

You will learn much more about the Debugger if you participate in this sample session. Execute the code file one time to see the program's output before attempting the sample session.

Loading the Debugger

As previously mentioned, the Debugger is not automatically loaded as part of the standard system, so you will need to load it into the computer. You can load the module in either of two ways:

- Execute it using the eXecute command (from the Main Command Level); the program installs itself.

- Add the DEBUGGER module to INITLIB, and re-boot the system; the program is then installed automatically.

Loading the Debugger with the eXecute command allows you to use it until you re-boot the system, at which time you will have to eXecute it again to use it. By adding the module to INITLIB, you give all users (who subsequently boot with this INITLIB file) access to the Debugger. You will not want to use this second method unless you want to give all users access to the Debugger.

Executing the Debugger

First, make sure that the ASM: disc is on-line or that the file is otherwise accessible. Then, from the Main Command Level, press the ⌈X⌉ key to initiate the eXecute command. The system will prompt you with this message:

```
Execute what file?
```

Respond by entering the specification of the DEBUGGER file; `ASM:DEBUGGER.` will work if you are loading the program from the original disc (remember to type a trailing period to suppress the `.CODE` suffix). The system then loads and executes the program, which installs itself permanently in memory.

Adding the Debugger to INITLIB

Use the Librarian (from the Main Command Level) to add this module to the INITLIB file. In general, the steps can be summarized as follows:

1. Make a back-up copy of INITLIB.

2. Edit the INITLIB file with the Librarian, adding the DEBUGGER module (supplied on the ASM: disc) to the file. The DEBUGGER can be anywhere *after* the modules named KEYS, BAT, CLOCK, and any CRT drivers (CRT, CRTB, etc.); however, it must be *before* the module named LAST. (Editing libraries is described in the Librarian chapter.)

3. Store the new library file (using the Keep command).

4. Remove the INITLIB file on your BOOT: disc, and add your new INITLIB file.

5. Re-boot the system.

After re-booting the system, the Debugger should be in memory.

A Note about Key Notations

Throughout this chapter, you will be shown which keys invoke certain Debugger functions. Since you may have one of various keyboards connected to your computer, each with a different set of keys, you will need to learn which key to press on your keyboard. Here are examples of keys used to invoke a few functions on the different keyboards[1].

Desired Function	HP 46020A/21A Key(s)	HP 98203B/C Key(s)	HP 98203A Key(s)
Pause	Break	PAUSE	PSE
Single Step	(System) f5	STEP	STEP
Slow Step	(System) CTRL–f5	CTRL–STEP	CTRL–STEP
Continue	(System) f4	CONTINUE	CONT

For instance, invoke the Pause function on a 46020/21 keyboard by pressing the **Break** key. On a 98203B/C keyboard, press the PAUSE key. With a 98203A keyboard, press the PSE key.

As another example, suppose that you want to invoke the Single-Step function. On 98203A, B and C keyboards, press the STEP key; the label is *on the key itself*. On a 46020/21 keyboard it will be the System key labeled f5 *on the key*, which is labeled **STEP** *on the screen* while in the System-key mode. (If you are not already in System-key mode, then you will need to press the System key before pressing f5). The same notation is used for the other System keys on the 46020/21 keyboard; the actual System key (i.e., f1 through f8) is not given in text; the label is given instead. You will need to make the association, which you can easily do by looking at the System-key labels while the Menu is being displayed (press the Menu key to toggle the Menu on and off). If you are not familiar with the System and Menu keys, read the discussion in the *Pascal User's Guide*.

The convention used in this manual is to show the 46020/21 keys first (followed by the equivalent 98203B/C key in parentheses). For instance, the Break (PAUSE) key invokes the Pause function: on the 46020/21, it is the Break key; on a 98203B/C keyboard, it is the PAUSE key. (The 98203A PSE key is not shown, because it is close enough to the PAUSE label that you should be able to easily make the connection.)

Is the Debugger Installed?

Before proceeding, you should verify that the Debugger is currently installed. On a 46020/21 keyboard, press Break (PAUSE) to pause the system. If a **p** is displayed in the lower, right-hand corner of the screen, then the Debugger is installed. Press **CONT** (CONTINUE) to resume operation.

If the Debugger is not installed, then pressing Break will do nothing.

Invoking the Debugger

The Debugger is called from the Main Command Level. When the Debugger is invoked, the system will then take steps to determine which program you want to debug. Before invoking the Debugger, let's look at how it determines which program to debug.

Specifying a Program

When the Debugger is invoked, the system will either look for a code file on its own or ask you for the code file's name, according to the following priorities. (If the Debugger is not installed, then the D command is identical to the eXecute command.)

1. If there is currently an object-code workfile, the file is automatically loaded into computer memory. If there is a source-code workfile (but not an object-code file), the system reports that it cannot open the file because it was not found.

2. If there is no workfile, the second check made is for the last file compiled since power-up. If present, that file is then loaded.

3. If neither such file exists, you are prompted for a file name.

If you plan to debug, edit, and recompile a program several times in a session, using a workfile may be the best alternative; you will not have to keep typing in the file name, because the current workfile is the automatic object of those subsystems.

For this session, we will set up a workfile. First, use the Filer's What command to see if there is already a workfile. If it happens to be the DEBUG.CODE file, you need do nothing more (before exiting the Filer). Otherwise, use the Get command to specify the example program as the workfile. Here is the prompt you will see:

```
Get what file?
```

Answer by entering the file specification of the example program. Type:

```
DEBUG Return or ENTER
```

The filer responds with something like this:

```
Source and code file loaded.
```

You may now Quit the Filer. Now press the D key while at the Main Command Level to invoke the Debugger.

Answering RESTART WITH DEBUGGER?

As mentioned earlier, this prompt is shown any time that a "user" program generates *but does not trap* an exception. Answering "Yes" to this question will also get you into the Debugger; you will effectively be at the same point as if you had used the D command. (If the Debugger is not currently installed, answering "Yes" will only re-execute the program.)

The Debugger Command Screen

You are now in the Debugger's command screen. This message indicates that the Debugger is ready for further instructions:

```
NOW AT START
>
```

The **d** shown at the lower, right-hand corner of the screen also indicates that you are currently in the Debugger.

The Debugger prompt is a **>**. When this screen and prompt are displayed, you can type Debugger commands on the line with the cursor. Enter each command by pressing the [Return], [ENTER], or [Select] ([EXECUTE]) key. Note that the **CONT** key resumes normal program execution. When execution of the program is complete, control returns to the Main Command Line.

Single-Stepping a Program

When the Debugger is at this starting point, it is ready to step through your Pascal program one statement at a time; this mode is called Single-Step Mode. (It can also do many other things, which will be discussed momentarily.) In the lower, right-hand corner of the screen, the Debugger also conveniently displays the program line number which contains the *next* statement to be executed (if the program was compiled with **$DEBUG ON$**. This line number corresponds to the line number given in the Compiler listing of the program.

For instance, when debugging our example program, line number 57 is initially displayed. This is the line that contains the Pascal statement that will be executed the next time you press the **STEP** key. Press the **STEP** key once and note that the line number changes to 58, which is the line number of the next statement to be executed.

Pressing **STEP** a second time results in no change in line number. This response is due to the fact that the Debugger steps through the program one *statement* at a time, not one *line* at a time. Pressing **STEP** a third time changes the line number to 59.

Slow Program Execution

The Debugger also allows you to execute a program at a rate of about two statements per second. Press [CTRL] - **STEP** to use this execution mode (Slow-Step Mode). Line numbers are flashed on the screen as each is encountered. You can return to Single-Step Mode by pressing the **STEP** key.

Returning to the Debugger Command Screen

You may have noticed that the Debugger prompt disappeared when you began stepping through the program. Instead, the Debugger displays the screen that will be used for the program's output so that you can see what the program is doing at each step of execution.

Note that any keys pressed while in this mode appear in the system's type-ahead buffer, not in a Debugger command line. This action allows you to type in responses to any input statements in the program as you would normally type them in. The program reads this buffer when an input statement is encountered and executed.

At some point in the program's execution you may want to return to the Debugger command screen to execute a command. To do so, press CTRL-Break (CTRL-PAUSE). The Debugger restores the last Debugger command screen, which is the one that you saw before you began single-stepping the program. You can then execute Debugger commands or return to the program screen by stepping through the program with the **STEP** key.

Toggling Between Screens

While in the Debugger command mode, you can also toggle between these displays (without changing modes) by pressing CTRL - **ALPHA**. For example, suppose you want to quickly check the program screen to see last line displayed by the program. You can do so (without getting out of the Debugger command mode) by pressing CTRL - **ALPHA**. When you've examined all you want on the program screen and are ready to return to the Debugger's command screen, press CTRL - **ALPHA** again.

Screen Dumps

While in the Debugger, you can dump the current contents of the alpha or graphics screens. Use either the **DMP A** (DUMP ALPHA) key or **DMP G** (DUMP GRAPHICS) key, or execute a DA or DG command.

Note that this feature is only allowed when running a program in the processor's "user" state[1]. It is not possible while executing programs in "supervisor" state. If attempted while disallowed, no dump is performed and the following message is displayed:

```
NOT ALLOWED NOW
```

---

[1] All user programs are executed in the "user" privilege state, while "Interrupt Service Routines" are executed in "supervisor" privilege state. See the *MC68000 User's Manual* for a more comprehensive description of these states.

A Look at the Queue

At this point, you may want to continue stepping through the program and noting the order of execution of lines. You can also get a log of all Pascal program lines executed thus far by the Debugger by executing the Queue command. (Actually, these are the line numbers of Pascal statements executed thus far.) Here is an example of the results of this command (assuming that we have only pressed the **STEP** key three times in our example):

```
>Q

206144^   69
206160^   69
206176^   68
206188^   67
START
```

The line numbers are shown in the right column. (The six-digit numbers in the left column, each followed by ^, are memory addresses for use when debugging Assembler language programs; you don't usually need to be concerned with them while debugging Pascal programs.)

Note that the line numbers in the queue are in *reverse order of execution*: the first line executed is at the bottom of the queue listing, the second is listed above the first, and so forth. Also note that the line at the top of the list has *not* yet been executed; it will be executed the next time the **STEP** key is pressed.

Note that Pascal line numbers will *only* be shown if the $DEBUG ON$ Compiler option was used.

Note also that when the question **RESTART WITH DEBUGGER?** is displayed after encountering an exception that was not trapped, you can get a listing of the queue by pressing [CTRL]-[Break] and then executing a Q command. You can also direct the Debugger to trap exceptions, as described in the Exception Trapping section of this chapter.

Displaying Data

Before showing how to use many of the more powerful Debugger features, let's look at some simple Display operations. Execute the following command:

```
>D 8+32
        +40
```

From this example, you can deduce that the literal numbers that you entered were interpreted as decimal integers and that the result was a signed decimal integer.

Note that you don't need to specify the D in commands that begin with non-alphabetic characters.

```
>8+32
        +40
```

Now execute this command:

```
>D -32768-32768
        -65536
```

From this result, you can see that the range of integers is at least 16 bits. In fact, it is 32 bits. The range is -2^{31} through $2^{31} - 1$ (or $-2\ 147\ 483\ 648$ through $2\ 147\ 483\ 647$).

Executing these commands might help to see this range of integers more clearly:

```
>D 127*256*256*256
+2130706432
>D 128*256*256*256

OVERFLOW
```

Note that *only* integer arithmetic operations are performed. For instance, division produces the integer portion of a quotient:

```
>5/3
        +1
```

Display Formats

Since the Debugger uses the processor's 32-bit registers for expression evaluation, most results are four-byte quantities and are formatted to reflect that fact. Here are two equivalent examples of using the default format of one signed (four-byte) INTEGER:

```
>D 255
        +255
>D 255:1I4
        +255
```

The format specifier is the ':1I4' appended to the literal number 255: the leading 1 indicates that one quantity is to be generated; the I indicates that the quantity is to be displayed as a signed, decimal Integer; the trailing 4 indicates that 4 bytes are to be formatted.

If the default format of one four-byte decimal integer is not what you'd like, you can explicitly specify another format. For example, the following command generates four one-byte binary numbers (the !'s indicate binary notation):

```
>D 1024+255:4B1
!00000000 !00000000 !00000100 !11111111
```

Here is an example of formatting the integer into four one-byte octal numbers (the %'s indicate octal notation):

```
>D 1024+255:4O1
%0 %0 %4 %377
```

Now specify that the number is to be formatted as one four-byte hexadecimal number with either of the equivalent commands (the $'s indicate hex notation):

```
>D 1024+255:1H4
$000004FF
>D 1024+255:H
$000004FF
```

The leading 1 and trailing 4 are the defaults assumed when these parameters are omitted.

This format specification directs the Debugger to display two bytes as a hex value:

```
>D 1024+255:H2
$0000
```

Note that 0's were displayed because the Debugger begins its display with the *most*-significant bits of the four-byte integer. Here is a more meaningful display format for the same data:

```
>D 1024+255:2H2
$0000 $04FF
```

It is also possible to display literal strings with the data you are formatting for the display. Either single or double quotes can be used to delimit the string. For example, this command gives a more descriptive display:

```
>D -7:'-7 in Hexadecimal = ',H4
-7 in Hexadecimal = $FFFFFFF9
```

You can also dis-assemble machine-language instructions by using the X format specifier. Here is an example:

```
>D $4E750000:X
RTS
```

This is usually only helpful while debugging Assembler-language programs. (Note that you must load module REVASM into memory with the P-load command from the Main Command Level in order to use this format.)

Another format specifier is the slash (/). When a "/" is encountered in a Display command, the display is continued on the next line.

```
>D 23+45:/,'RESULT = ',I4/

RESULT =        +68

>
```

Input Formats

The !, %, and $ symbols preceding numbers in the above examples were used to indicate the base of the numbers displayed on the screen. Similarly, you can use them with literal numbers input in the command. This feature allows number-base conversions.

For example, suppose that you want to convert the binary number 11001010 to its decimal representation. Here is a sample command:

```
>D !11001010:I
        +202
```

To display the number in hex, execute this command:

```
>D !00110110:H
$00000036
```

Changing the Default Display Format

The default format can be changed by giving an F (Format) command. For example, the following command changes the default to ':1H4', which instructs the Debugger to take 4 bytes and display them as one four-byte Hexadecimal value:

```
FH
```

This command sets the default format to Octal (':1O4'):

```
FO
```

This command changes the default format to ':1U4', which directs the Debugger to display 1 four-byte Unsigned decimal integer.

```
FU
```

This command sets the default format back to signed decimal Integer (1I4):

```
FI
```

Now that you've had an introduction to the Display commands, let's look at some more powerful commands.

Controlling Execution with Breakpoints

A breakpoint is a point in the program where you want execution to be temporarily halted. With a Pascal program, the point will be at a program line. Thus, when the Debugger is executing a program and encounters a breakpoint, it halts just *before* executing the program line.

Setting Breakpoints

To set a breakpoint, use the BS command. Specify the location as an integer which follows the letters "BS", separated by a space. For example, to set a breakpoint at Pascal program line 74, enter the following command:

```
BS 74
```

Press **CONT** (**CONTINUE**) and the program begins executing again. When it encounters line 74, it pauses before executing the line and displays the message:

```
NOW AT LINE 74
```

The Debugger then prompts you for another command. At this point, you can do any of the following:

- Step through the program one line at a time (if it was compiled with $DEBUG ON$)

- Execute other Debugger commands (such as examine memory or register contents)

- Continue the program

Once the program has finished execution, all breakpoints are automatically de-activated. You will have to explicitly re-activate them, as described in a subsequent section.

Up to nine such breakpoints may be defined at one time. Most breakpoints remain in effect until cleared or de-activated.

The Count Option

An optional count can be included by adding an integer after the location. The count instructs the Debugger to stop when it reaches the location the indicated number of times. For example, enter the following command:

```
BS 31 3
```

This particular command instructs the Debugger to halt the program immediately before the third execution of line 31. Press **CONT**, and the program executes until line 31 is reached the third time and then halts. Note that this type of breakpoint is automatically cleared when encountered the specified number of times.

Breakpoints with Commands

Another form of the BS command is the "BS" and the location number followed by a Debugger command string enclosed in quotes. The command string is one or more legal Debugger commands (separated by semi-colons). These commands are immediately executed when the location is encountered. The Debugger automatically continues program execution after executing the command string. Here is an example that will provide a visual record of how many times line 37 was executed:

```
BS 37 "D 'LINE 37 REACHED.'"
```

Of course, you will need to get back into the Debugger command screen to see the results of this breakpoint being encountered. The D (Display) command is explained in detail later.

You can pause the program by making the last command in the string a question mark. This command directs the Debugger to pause and wait for input from the keyboard. For example, enter the following command breakpoint:

```
BS 59 "D PC; ?"
```

The Debugger stops at line 59, displays the Program Counter, and waits for input.

Here is another example of using a breakpoint with a command:

```
BS 59 'IF 1=1; D "1=1"; ELSE; D "1<>1";?; END'
```

The relational expression following the IF command, in this case 1=1, is first evaluated. If it is true, then the command(s) between the IF and the ELSE are executed. If it is false, then the command(s) between the ELSE and END are executed. This type of command is useful for purposes like checking the value of a variable and then pausing if its value is out of an expected range. The IF, ELSE, and END commands are further explained in the reference section. Checking the value of variables is explained later in this tutorial.

Deactivating Breakpoints

The BD command deactivates breakpoints. If a line number is included, the breakpoint is deactivated for that line number. For instance, the following command deactivates the breakpoint at line 41:

```
BD 41
```

If no line number is included, all breakpoints are disabled. For example, this command disables all breakpoints:

```
BD
```

Displaying the Breakpoint Table

The B command displays the breakpoint table or the breakpoint at the specified line number. Execute the following command:

```
B
```

and you'll see a display similar to the following:

```
>B
BREAK POINTS
------------
A          74          0
A          37 D 'LINE 37 REACHED'
A          59 D 'IF 1=1;D "1=1";ELSE;D "1<>1";?;END
```

The first character on each line of the table is either "A" for active, or "D" for deactivated. The second parameter is the line number of the breakpoint. If the third entry in the table is a positive number, then a count option is in effect for the breakpoint (execution will pause when the Debugger reaches the line that number of times). If the third entry is a command string, then that command is executed each time the line is encountered. If it is a "0", then it is a normal breakpoint (i.e., no count nor command was specified with the breakpoint).

Reactivating Breakpoints

The BA command reactivates disabled breakpoints. If the line number is included, the breakpoint is reactivated for that line number, otherwise, all breakpoints are affected. For example, the following command reactivates the breakpoint that was deactivated in the example above:

```
BA 41
```

Try the B command to see the table again.

When a program runs to completion and is then restarted (by pressing the ⏹D key), the breakpoints are still there; they are just deactivated. Use the BA command to reactivate some or all breakpoints.

Clearing Breakpoints

The BC command clears breakpoints by removing them from the table. If a line number is included, the breakpoint is removed *only* for that line; otherwise, all breakpoints are cleared. Enter the following command to remove only the breakpoint at line 41:

```
BC 41
```

The Pause Function and Breakpoints

If the Debugger is not installed, the ⌈Break⌉ (⌈PAUSE⌉) key is a no-op. The rest of this discussion assumes that the Debugger is installed.

While *not* in the Debugger command mode, pressing ⌈Break⌉ effectively halts any program at the current execution point. (Note that this key may not pause the program on a line boundary like the **STEP** key does.)

While *in* the Debugger command mode, however, pressing ⌈Break⌉ returns you to the user program display and pauses the program at the current execution point. Press continue to finish program execution.

If you press ⌈Break⌉ after encountering an active breakpoint, it will also get you to the program's display. However, if you pause exactly on a currently active breakpoint (but before encountering it), pressing ⌈Break⌉ will not get you into the program's display. You will have to press ⌈Break⌉ again to cause the breakpoint to take immediate effect. **CONT** will then work as expected.

Executing a Number of Statements

Go commands set a tenth temporary breakpoint. They are one-time commands to pause execution before a specified program instruction.

The G command tells the Debugger to Go. If you include a number after the "G", that number of statements is executed, after which the Debugger halts and waits for another command. For example, this command tells the Debugger to Go 8 statements:

```
G 8
```

If no number is given, the remaining instructions are executed (same as pressing **CONT**).

The GF (Go and Flash) command is the same as the G command except execution is slowed and line numbers are flashed in the lower right corner of the screen.

The GT (Go 'Til) command is the same as Go except a location is specified rather than a count. For example, this command tells the Debugger to Go 'Til line 39 is reached:

```
GT 39
```

Another form of this command tells the Debugger to Go 'Til the location is reached a number of times. For example, the following command tells the Debugger to stop before line 41 is executed the third time.

```
GT 41 3
```

The GT statement also allows the command string option. For example, this command directs the Debugger to do the following: execute the program until line 42 is reached, then display the Program Counter and await further instructions.

```
GT 42 "D PC; ?"
```

The GTF (Go 'Til and Flash) command is the same as GT except execution is slowed and line numbers are flashed in the lower right corner of the CRT.

Tracing Program Flow through Procedures

You can also halt execution of a program as it enters and exits procedures. For instance, suppose that you want to halt the program when the current procedure is exited. To do that, execute the PX (Procedure eXit) command:

```
PX
```

Execution will be halted after the procedure is exited (i.e., after the last line of this procedure is executed, but before the subsequent program line is executed). For instance, executing this command while in Level_3 results in this display:

```
>PX
PROC EXITED
```

The Debugger shows that the next line to be executed is line 31.

To halt the program at the point that the current procedure (or main program) calls another procedure, use the PN (Procedure Next) command:

```
PN
```

When the next procedure is encountered, the Debugger reports this message:

```
NEXT PROC
```

and the program is halted before executing the first executable line of the procedure. If the current procedure is exited before another is called, the Debugger reports this message:

```
PROC EXITED
```

and the program is halted before the next executable line of the calling procedure is executed.

A Look at the Stack Frame

Another handy feature to use while walking though the program on a procedure basis is the SF (Stack Frame) command. Here is an example display of this command:

```
>SF
           46  LEVEL_1
PROC ADDRESS      -403316^
CALLED FROM       -402718^
         LINE           55
```

The first line of the display shows the first (executable) line of the program next to the procedure's name. The second line shows the memory address of the procedure (which is not important while debugging Pascal programs). The third line shows the address of the procedure. The fourth line shows the number of the line from which this procedure was called.

Examining Variables

Without the ability to check the value of program variables, debugging a program could become more tedious than it already is. Rest assured that this Debugger does allow you to look at the contents of any variable in computer memory. However, in order to check the contents of program variables, you will need to know two important facts: where they are in memory, and how to format them into an understandable form.

To see where a variable is stored in memory, it is necessary to look at the Compiler listing. Each variable has a negative integer printed next to it on the listing. This negative value is the offset (in bytes) from the base address where the variables are located. The base address for a procedure's local variables is the current stack frame pointer (SF); the main program's variables have a base address offset which is the value of the program name (here XYZ) from A5.

That's why it's helpful if, when writing the program, you declare each variable on a separate line so that an offset will be printed on the listing for each variable. Alternatively, you can use the $TABLES$ Compiler option to get a printout which tells all about each data type and variable. This option is explained further under "Formats for Structured Variables".

To format the variable's value in memory, you will need to use the Display command. Let's go back to the example program and let it finish by clearing the breakpoints using "BC" and then press **CONT**. Restart the program and then execute **GT 60** command to Go Til line number 60.

To see the value of the local variable i that is declared in the procedure called Level_1, look at the Compiler listing (line 14) to see that it has an offset of −4. This is an offset from the stack frame pointer (SF) of that procedure. Subtract 4 from the stack frame pointer, and use "^" after the expression to indicate you want the contents of the memory location *referenced by* the value of the expression in parentheses. Enter the following command:

```
>D (SF-4)^:'  X = ',I
   X = 0
```

To see the value of y, execute:

```
>D (SF-8)^:'  Y = ',I
   Y = 0
```

And to see the value of z, execute:

```
>D (SF-12)^:'  Z = ',I
   Z = 0
```

You may also specify that all three integer variables be displayed at the same time by executing this command:

```
>D (SF-12)^:3
0      0      0
```

The display will show the three integer variables separated with spaces. The variable with the offset of −12 will be the first one displayed, the one with the offset of −8 is second, then the third one has offset −4.

When looking for local variable values, be sure that you have stopped the program in the procedure that defines the variables. Each procedure that is called has a stack frame created for it even if there are no local variables. If you have stopped the program in a procedure which is contained inside of or called by another procedure, you can use the walk commands to get to the stack frames of the outer level procedures (see "Static and Dynamic Links").

The global variables in the main program or globals declared in modules are located at offsets from their specific global bases. The global areas each have a symbol associated with them. The symbol has a value which is the offset or distance from (A5). So when you reference global variables, add the program or module name to A5 and then subtract the offset for the particular variable location.

For example, if you wanted to see the value of the variable x in the main program (here it is named XYZ), use this command:

```
D (A5+XYZ-4)^
```

To see the value of Y, execute:

```
D (A5+XYZ-8)^
```

To see the value of the two character variables in ch1 and ch2 (of program DEBUG), it is necessary to specify a format, because the default format is integer. To see the variables ch1 and ch2, execute this command:

```
D (A5+XYZ-26)^:2A1
```

The format specifies that 2 ASCII values are to be displayed, each having 1 byte. They are located at an offset of -26 from the value of symbol XYZ, relative to A5.

The processor registers that can have their values displayed are listed below:

```
A0..A7      (the Address registers)
AA          (All Address registers)
D0..D7      (the Data registers)
DD          (all Data registers)
PC          (the Program Counter)
SR          (the Status register)
```

To display the numeric values of the contents of address register A0 and the Program Counter, execute this command:

```
D A0 PC
```

To display the numeric value at the location *referenced* by the the Program Counter (i.e., whose address is stored in the PC), execute the following command:

```
D PC^
```

To display the value at the location *referenced* by the Program Counter, interpreting it as an Assembler language instruction, execute this command (remember that module REVASM must be P-loaded to use this format):

```
D PC^:X
```

The Debugger symbols and corresponding definitions are as follows:

```
LN (Line Number)
EC (Escape Code)
IO (I/O result code)
GB (the Global variable Base)
RB (the code Relocation Base)
SF (the current Stack Frame pointer)
```

Examining Consecutive Memory Locations

The Open command is like the Display command except the address is displayed with the value and you are prompted to press either the up-arrow key or the down-arrow key. This causes the address value to increment or decrement depending on the key choice. The adjustment is 1 byte with the OB command, 2 bytes with the OW command and 4 bytes with the OL command. When you have seen enough, press ⎡Return⎤, ⎡ENTER⎤, or ⎡Select⎤ (⎡EXECUTE⎤) to terminate Open mode and return to the Debugger command mode. For example, to see the hex values which are the machine codes for the current program, use this command:

```
FH
```

(See the Default Formats section for more details.)

To examine the (16-bit) word pointed to by the current contents of the Program Counter, use this command:

```
OW PC^
```

The > to the right of the display prompts for an up-arrow key (⎡^⎤) or down-arrow key (⎡v⎤). To see the next word in memory, press the up-arrow key. Continue until you have seen enough. Press ⎡Return⎤ or ⎡ENTER⎤ to exit the Open command.

Formats for Structured Variables

There is a mechanism for displaying non sequential values also. It is necessary to specify one memory location to set the memory pointer. Then by using special symbols, you can alter the value in the memory pointer. You can also display the value of the memory pointer. All these symbols are part of the format and are typed following the location specification and a colon (:).

```
"*" is the value of the memory pointer

"<" preceded by a number, decrements the value of the memory
    pointer by the number

">" preceded by a number, increments the value of the memory
    pointer by the number

"^" causes the memory pointer to take the value at the
    location indicated by the current pointer
```

These mechanisms make it possible to examine different fields of structured variables.

First, a note about structured variables. When space is allocated for a structured variable, the number of bytes needed is determined and given to the variable. The individual elements of the structure are then assigned space at ascending locations. For example, if you had the following Pascal record, 14 bytes are needed to store the whole record:

```
Pasc_Rec = RECORD
              x : INTEGER;
              y : INTEGER;
              ch1 : CHAR;
              ch2 : CHAR;
              pointer : ^Pasc_Rec;
           END;
```

If a variable of this type is the first variable for a procedure, then the record would occupy the first 14 bytes below the stack frame pointer (SF-14)^. The elements in the record would be at positive offsets from this location. Variable x would have an offset of 0 (SF-14)^; y has an offset of 4 (SF-14+4)^; Ch1 has an offset of 8 (SF-14+8)^; etc. This information is easily obtainable when the $TABLES$ Compiler directive is used, and a Compiler listing is generated.

The following drawing illustrates the structure of the RECORD variable in memory.
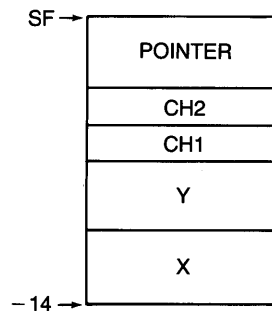
Rather than displaying the values of the record individually, you can use the following Debugger command:

```
D (SF-14)^:I4,4>,2A1,^,4>,I4,*
```

This command tells the Debugger to go to the memory location 14 bytes below the Stack Frame pointer (the bottom of the record), display the four-byte integer (x), go up 4 bytes and display the 2 Alpha characters, assume the value that is stored after the characters (the pointer field), then go up 4 bytes in the new record and display the four-byte integer (y), and then display the current location. Notice that the Debugger display pointer is left at the subsequent locations after the particular displays are made. In other words, after the display of (x), it is only necessary to move 4 bytes rather than 8, to position the display pointer to the character variables.

Changing Memory Contents

The ability to change the values in memory is, among other things, the ability to get a program back on the right track. In one Debugger session, you can detect several problems with a program without having to stop, edit and recompile the program for each one. Simply change the values of the variables that are causing the problem. To change the values of variables in a Pascal program, use the Open commands. Variables are referenced the same way they are with the Display command.

The Open commands are as follows:

- OB - for byte values.
- OW - for word values.
- OL - for long word (four-byte) values.

Suppose you want to change the value of a variable to 8; assume that it is local to the current procedure, that it is an integer variable, and that it has an offset of −4 from the procedure's stack frame pointer (SF). It is necessary to use the OL form of the Open command, since integers are 4 bytes long. Execute the following command:

```
OL (SF-4)^ 8
```

As another example, suppose you want to change the value of the global (main program's) variable ch1 to "x". Because characters only use 1 byte of storage, use the OB form of the command.

```
OB (GB-25)^ "x"
```

By changing the values of those variables, the sequence of execution is drastically altered.

Static and Dynamic Links

Each time a procedure is called in a Pascal program, a new stack frame is created. This stack frame contains all the local variables in the procedure as well as the procedure's static and dynamic links[1].

The Debugger contains a mechanism for following these links. It is the Walk command. The Walk command takes three forms:

- WS - follows the static link back one step.

- WD - follows the dynamic link back one step.

- WR - resets to the current stack frame.

There are no options or parameters. These commands in no way affect or influence program execution.

Restart the Debugger by pressing the [Stop] key and the [D] key. Set a breakpoint on line 31 for the third execution of the procedure Level_3.

    BS 31 3

Press [Return] or [ENTER], and then press **CONT**. The program will stop the third time line 31 is reached.

The sequence of calls is as follows:

    Program XYZ

    Procedure Level_1

    Procedure Level_2b

    Procedure Level_2a

    Procedure Level_3

    Procedure Level_3

    Procedure Level_3

Give six successive WD (Walk Dynamic) commands and you'll get the above information presented in reverse order. The information displayed for each WD command is the stack frame information for the current procedure and then the same for the calling procedure. The stack frame pointer is updated to point to the calling procedure's stack frame. You can look at those variables and the links stored in that stack frame. Consecutive WD commands walk us back through the entire calling sequence. We can stop anywhere along this path and examine the variables in a procedure's stack frame.

---

[1] Static and dynamic links are described in detail in the section of the Compiler chapter called How Pascal Programs Use the Stack.

To return to the stack frame for Level_3 where you stopped the program, execute a Walk Reset command:

    WR

This command resets the Debugger stack frame pointer variable.

You can also walk the static link. This gives you the ability to examine variables whose scope statically (textually) includes the current procedure. Execute:

    WS

This command brings us to the Stack Frame for Level_2a which contains the variables i, x, and y.

Use the Display command to examine the value of i.

    D (SF-4)^

The value of i is displayed.

The value of i is only affected by successive executions of Level_3. If Level_3 had local variables, they would display different values in each stack frame. However, only one copy of the variable i exists in the one stack frame for procedure Level_2a. The value of i is as it was when we stopped program execution during the third invocation of Level_3. That value is 3.

Exception Trapping

It is possible to stop execution of a program at an exception to normal processing. Normally, an escape is made by the system and successive recovery mechanisms allow termination of the program. At the time of termination, the system displays the escape code and the line number in the outer level recovery (if the program was compiled with $DEBUG ON$). The escape code is valid information, but the line number may not be the location of the error. By re-executing the program with a trap set for the exception, we can stop execution at the point of the error, have the actual line number of the error displayed, and examine variables for the problem.

There are three commands for exception trapping. We can trap selected escape codes with the Escape Trap instruction. The following command directs the Debugger to trap only escape code 100.

    ET 100

When escape code 100 is encountered, control is returned to the Debugger and the following message is displayed on the screen:

    -EXCEPTION-
    ESCAPE CODE 100
    SR=$0000 PC=   -207532 LINE   +12

We can stop at all *except* selected escape codes with the Escape Trap Not instruction. This command directs the Debugger to trap every escape code except 100.

```
ETN 100
```

Not specifying an escape code causes the command to work for every escape code. This command directs the Debugger to trap *all* escape codes.

```
ET
```

This command doesn't trap any exceptions.

```
ETN
```

When the exception occurs, execution stops and control is transferred to the Debugger. At that point, you can examine the state of the program.

When the Debugger is initiated, the default escape trapping command is the following:

```
ETN 0 -20
```

These are the escape codes for normal termination and the [Stop] key. The Debugger will trap all escape codes except those.

The third type of escape trap command allows you to execute command(s) when the escape is detected. Here is an example:

```
ETC 'D "ESCAPE HAS OCCURRED";?'
```

This command displays its message and then halts the program, awaiting further Debugger commands.

Generating Escapes

With the Debugger, you can also generate escapes. For instance, this command generates an ESCAPE(10) at the current point in the program.

```
>EC 10
```

The result of this command is the same as if the program had encountered the escape at the current location. If you have an ET command currently defined for the escape code, the Debugger will trap it also.

A Note about Assembly Language Programs

All of the Debugger commands apply when debugging an Assembler language program as well. The difference is that the location specification is given as an address and not a line number. An address is specified with a "^" appended to the location specifier. For example, the following command says to Go Til the address FFFF 1432 is encountered:

    GT FFFF1423^

The Debugger knows about symbols which have been DEFed. The entry points into assembly modules, programs, and procedures should have been defined (with DEF). You can specify an address in an assembly routine by specifying an offset from the routine's entry point. The offset in the routine can be found on the Assembler output. For example, the following (equivalent) commands direct the Debugger to Go Til encountering the the address 16 decimal (10 hex) memory locations past the entry point into "routine":

    GT (routine+16)^

or:

    GT (routine+$10)^

Note that (routine+10)^ should be that start of a MC68xxx opcode for the Debugger to pause there; if the address points at the middle of an instruction (e.g. an operand), or to data, the Debugger will not pause there.

Read about the particulars of each command in the subsequent Command Reference section.

Debugger Named Reboot

Named reboot is a cabability added to the Debugger in revision 3.22. It gives you a way to specify a system to reboot using extensions to the **sb** command.

The format of the system boot (**sb**) command of the debugger is show below. It has been expanded to offer reboot capability based on the parameters you provide. If no parameters are given, the pre-3.22 usage will not be affected. **sb** must be in lower-case letters (use the Shift key to generate lower-case letters in the Debugger), and there must be a space preceeding each parameter.

```
sb [SYSTEMNAME|* [MSUS|* [LANID|*]]]
```

NAME

The SYSTEMNAME parameter is one to ten ASCII characters long and should contain the name of a boot file (boot file names seen by the boot ROM at power up can be displayed by pressing any key after the boot ROM has seen the keyboard). The name is a string and must be within quotes.

If the SYSTEMNAME parameter is given as an *, then the currently booted systems's name will be used.

MSUS

MSUS is an acronym for Mass Storage Unit Specifier, although current systems can also boot from devices such as ROM or Local Area Networks (LAN) which are not thought of as mass storage devices.

The MSUS string may be one of three forms:

- An * will cause the currently booted system's MSUS to be used.

- A file system unit specifier. Unit specifiers should have no trailing colon—*#11* is correct, *#11:* is incorrect. Only unit specifiers are allowed: a volume name would be an error.

- An 8 hex-digit number (32 bits) which will replace the boot ROM supplied 32 bit MSUS. This MSUS contains four fields of 2 hex digits (8 bits) each. If the MSUS parameter were coded *$11000700*, it would mean:

Device ID	Unit Number	Select Code	Bus Address
$11 (CS80 device)	$00 (Volume 0, Unit 0)	$07	$00

Listed below are the parameter values for MSUS.

ID (Device ID)	Value (defined as of December 1988)
$00	Series 200 internal 5.25in mini-floppy
$04	9895 8in floppy or 913x 5.25in micro-winchester (HP-IB)
$05	82900 series 5.25in mini-floppy (HP-IB)
$06	9885 8in floppy (GPIO)
$07	913xA 5 megabyte 5.25in micro-winchester (HP-IB)
$08	913xB 10 megabyte 5.25in micro-winchester (HP-IB)
$09	913xC 15 megabyte 5.25in micro-winchester (HP-IB)
$0A	7905 hard disc (HP-IB)
$0B	7906 hard disc (HP-IB)
$0C	7920 hard disc (HP-IB)
$0D	7925 hard disc (HP-IB)
$0E	SCSI Direct Access Devices
$10	CS/80 and SS/80 devices with 256-byte blocks (HPIB)
$11	All other CS/80 and SS/80 devices (HP-IB)
$14	EPROM card (HP98255)
$16	BUBBLES (HP98259)
$E0	ROM (no other MSUS fields apply)
$E1	SRM
$E2	LAN (only the MSUS select code field applies)

Note

The presence of a device type in the above list does not imply Pascal Workstation support for the device, nor does it imply the support by all boot ROM revisions for the device.

UN (Unit Nunber)	Value (dependent on ID value)
For ID $00	UN 00 is the right-hand drive, UN 01 is the left-hand drive
For ID $10 and $11	First digit of UN is a volume number and the second digit is the unit number
For ID $14	This field indicates the device's position relative to other EPROM devices. For instance, *01* is the lowest EPROM device, *02* is the second lowest EPROM device, etc.
For ID $E0, $E2, and $16	No unit number exists, therefore this number is treated as a *don't care*.
For all other ID's	UN is encoded with the unit number associated with the physical device.

SC (Select Code)	Value
SC (Select Code)	The select code associated with the physical device is placed here for all Device ID's except $00, $E0 and $14. The internal HP-IB is coded as $07.

BA (Bus Address)	Value
BA (Bus Address)	For Device ID's $E1, $07, or $0E, this field contains the bus address associated with that physical device.

LANID

LANID is required only when the Device ID in the MSUS indicates a LAN ($E2). It contains the 12 digit HEX string that identifies the boot system across the LAN.

Examples

Below are examples using the sb command. If the debugger finds an error when parsing the parameters, it displays an error message and cancels the reboot.

sb	reboot, ROM will perform an UNATTENDED boot
sb *	reboot the current system
sb * *	reboot the current system
sb * * *	reboot the current system
sb 'SYSHPUX'	reboot system SYSHPUX from the current boot media
sb 'SYSP3.22' $E1081500	reboot system SYSP3.22 from an SRM
sb 'SYSP3.22' #11	reboot system SYSP3.22 from the mass storage device specified at unit number 11
sb * #11	reboot same system name from the mass storage device specified at unit number 11
sb * * '080009000986'	reboot same system name from different host

Debugger Keyboard

This section describes the key definitions while in the Debugger. Note that once you are in the Debugger there are two modes: Command Mode and Step Mode.

A Note about Key Notations

Throughout this section, you will be shown which keys invoke certain Debugger functions. Since you may have one of various keyboards connected to your computer, each with a different set of keys, you will need to learn which key to press on your keyboard. Here are examples of keys used to invoke a few functions on the different keyboards.

Desired Function	HP 46020A/21A Key(s)	HP 98203B/C Key(s)	HP 98203A Key(s)
Pause	Break	PAUSE	PSE
Single Step	(System) f5	STEP	STEP
Slow Step	(System) CTRL—f5	CTRL—STEP	CTRL—STEP
Continue	(System) f4	CONTINUE	CONT

For instance, invoke the Pause function on a 46020/21 keyboard by pressing the **Break** key. On a 98203B/C keyboard, press the PAUSE key. With a 98203A keyboard, press the **PSE** key.

As another example, suppose that you want to invoke the Single-Step function. On both 98203A, B and C keyboards, press the STEP key; the label is *on the key itself*. On a 46020/21 keyboard it will be the System key labeled f5 *on the key*, which is labeled **STEP** *on the screen* while in the System-key mode. (If you are not already in System-key mode, then you will need to press the System key before pressing f5). The same notation is used for the other System keys on the 46020/21 keyboard (i.e., f1 through f8): the actual System key is not given in text; the label is given instead. You will need to make the association, which you can easily do by looking at the System-key labels while the Menu is being displayed (press the Menu key to toggle the Menu on and off). If you are not familiar with the System and Menu keys, read the discussion in the *Pascal User's Guide*.

The convention used in this manual is to show the 46020/21 keys first (followed by the equivalent 98203B key in parentheses). For instance, the Break (PAUSE) key invokes the Pause function: on the 46020/21, it is the Break key; on a 98203B/C keyboard, it is the PAUSE key. (The 98203A PSE key is not shown, because it is close enough to the PAUSE label that you should be able to easily make the connection.)

Is the Debugger Installed?

Before proceeding, you should verify that the Debugger is currently installed. Press `Break` (`PAUSE`) to pause the system. If a p is displayed in the lower, right-hand corner of the screen, then the Debugger is installed. Press **CONT** (`CONTINUE`) to resume operation.

If the Debugger is not installed, then pressing `Break` will do nothing.

Calling the Debugger from the Main Command Level

`D`   From the Main Command Level, pressing the `D` key calls the Debugger (if installed).

Step Modes

Here are the available operations and key definitions while in the Debugger Single-Step and Slow-Step Mode.

Getting into the Step Modes

STEP   Causes the program to halt on the next line number; or, if already halted, execute one Pascal statement. (This key gets you into the Single-Step Mode.)

`CTRL`-**STEP**   Causes program execution to be slowed (to about 2 statements per second) and line numbers displayed. (This key gets you into the Slow-Step Mode.)

Controlling Program Execution

`Break`   Program execution is paused. Note that the type-ahead buffer is still active and
(**PAUSE**)   immediate-execute keys still function (e.g,. **DMP A**).

`Stop`   Stops program execution.

Getting into Command Mode

`CTRL`-`Break`   This key provides immediate entry into Debugger Command Mode. (Do not use
(`CTRL`-   `Shift`-`Reset` to do this, unless `CTRL`-`Break` does not work, and you must enter
PAUSE)   Command Mode. `Shift`-`Reset` may corrupt the system, and require rebooting to restore it.

Returning to the Main Command Level

CONT   Causes program execution to resume with Step mode cancelled.
(**CONTINUE**)

Command Mode

Here is a description of available operations and key definitions while in the Debugger Command Mode. If it is not installed, the command is identical to the eXecute command.

Entering Commands

Alphanumeric Keys	Used to enter Debugger commands. The characters generated are uppercase; you must use [Shift] to produce lowercase characters. [Caps] ([CAPS LOCK]) is disabled in the Debugger.
[Return] or [Enter]	Terminates input and initiates execution of the command.
[Select] (**EXECUTE**)	Terminates input and initiates execution of the command.
[CTRL]-**ALPHA**	Alternates between the Debugger command screen and System screen ([CTRL]-**EXEC** on the 98203A keyboard).
[Caps]	Disabled (keyboard is always in CAPS mode).
[Shift]	With numeric-pad keys, produces special characters (only 46020/21 keyboards).
[CTRL]	With alphanumeric and numeric-pad keys, allows entry of ASCII control characters.
[Back space]	Back spaces the cursor and blanks one character (If the cursor is at the extreme left, this key is a no-op).
[Clear line] or [Delete line] ([Clear line] or [Delete line])	Clears the input line.
RECALL	Clears the input line and recalls the last executed line.
[Insert char] ([Insert char])	Inserts one (1) blank character at the cursor position (does *not* switch to an "insert mode," as there is none).
[Delete char] ([Delete char])	Deletes the character at the cursor position.
CLR->END	Deletes all characters to the right of the cursor.
k0, [f1] thru [f8], and **k9** (**k0** thru **k9**)	Typing-aid keys (explained under K commands).
Knob	Same as left/right arrow keys.
[Shift]-Knob	Same as up/down arrow keys.
Left arrow and **Right arrow**	Move the cursor in the corresponding direction.
[▲] and [▼]	Have meaning only with the Open commands (OL, OW, OB).

Screen Control

[Clear display] ([Clear display])	Clears the alpha raster. In Step Modes, this key clears the System screen; in Command mode, it clears the Debugger Command screen.
ALPHA	Turns on the alpha raster and turns off the graphics raster ([Shift]-**RCL** on the 98203A keyboard). Disabled on bitmap displays.
DMP A (**DUMP ALPHA**)	Performs a DUMP ALPHA function (the current alpha raster, or bitmap display, is sent to the PRINTER: volume). ([SHIFT]-[INS C] on the 98203A keyboard).
[CTRL]-**ALPHA**	Alternates between the Debugger and System screen images ([CTRL]-**EXEC** on the 98203A keyboard).
GRAPH (**GRAPHICS**)	Turns on the graphics raster and turns off the alpha raster. ([Shift]-[Insert line] on the 98203A keyboard.) Disabled on bitmap displays.
DMP G (**DUMP GRAPHICS**)	Performs a DUMP GRAPHICS function (sends the current graphics raster, or bitmap display, to the PRINTER: volume). ([Shift]-[Delete char] on the 98203A keyboard.)

Controlling Program Execution

[Break] (**PAUSE**)	Program execution is paused. Note that the type-ahead buffer is still active and immediate-execute keys still function (e.g,. **DMP A**).
[Stop]	Stops program execution.

Getting into Step Mode

STEP	Causes the program to continue executing until the next line number is encountered (i.e., gets you into Single-Step Mode).
[CTRL]-**STEP**	Causes the program to continue executing slowly, displaying line numbers as they are encountered (i.e., gets you into Slow-Step Mode).

Returning to the Main Command Level

CONT (**CONTINUE**)	Causes program execution to resume with Command mode cancelled.

Debugger Command Summary

This section briefly summarizes the Debugger commands for quick reference purposes. A more complete description of each command is presented in the following Command Reference section.

Breakpoint Commands

BS	Sets a breakpoint at the specified location.
BD	Disables (but does not remove) breakpoint(s).
BA	Activates disabled breakpoint(s).
BC	Clears breakpoint(s).
B	Displays the breakpoint table.

Call Command

CALL	Calls the machine language routine at the specified memory address.

Display Commands

D	Displays the specified object(s). Objects can be specified immediately, directly, or indirectly. Formats describe the internal representation of the data.
TD	Displays the command string which is defined by the softkey **k4**.
TD I	Restores the initial command string to **k4**.

Dump Commands

DA	Performs the DUMP ALPHA function.
DG	Performs the DUMP GRAPHICS function.

Escape Code Commands

EC	Generates the specified escape.
ET	Sets up escape trapping of specified escape codes; Debugger halts when an escape is executed.
ETC	Sets up escape trapping of all codes; Debugger executes the specified command when an escape is executed.
ETN	Sets up escape trapping of all codes *except* those specified; Debugger executes the specified command when an escape is executed.

Format Commands

FB   Sets the default display format to Binary.

FH   Sets the default display format to Hexadecimal.

FI   Sets the default display format to signed Integer.

FO   Sets the default display format to Octal.

FU   Sets the default display format to Unsigned integer.

Go Commands

G   Causes execution to resume (same as **CONTINUE**).

GT   Causes execution to resume until specified location is encountered.

GTF or **GFT** Same as GT except that execution is slowed and the line numbers are flashed in the lower right-hand corner of the screen.

IF, ELSE, and END Commands

IF   Allows conditional execution of subsequent commands based on the result of evaluating the specified expression.

ELSE   Delimits the commands that will be executed when the IF condition is FALSE.

END   Ends the IF command.

Open Memory Commands

OB, **OL**, and **OW**   Used to display (and optionally alter) the values of memory locations.

Procedure Commands

PN   Halts program execution when the next procedure is called (or when the current one is exited, whichever occurs first).

PX, or **P** Halts program execution when the current procedure is exited.

Queue Commands

Q   Displays the Queue, which is a record of which line numbers were executed (or PC values of instructions executed).

QE   Ends recording of line number values in the Queue.

QS   Starts the recording of information in the Queue.

Register Operations

A0..A7,   Display or assign values to the corresponding processor register(s).
D0..D7,
PC, SP, US,
SR

Softkey Commands

k0 .. k9   Defines the command string to be displayed when the softkey is pressed (while in the Debugger).

System Boot Command

sb   The system boot command puts the computer in the power-up state for re-booting. (The command must be typed in lowercase letters.)

Trace Commands

T   Causes the specified number of instructions to be executed, each followed by an implicit TD command.

TQ   Same as the T command except that the TD command is executed only after the last instruction.

TT   Same as TQ except that a location is specified rather than a count.

Walk Procedure Links Commands

WD   The Stack Frame pointer (SF) is moved to the stack frame of the calling procedure.

WS   The SF is moved to the stack frame of the nesting procedure.

WR   The SF is returned to the current stack frame.

Debugger Command Reference

This section contains a formal description of syntax and semantics for each Debugger command.

Debugger Expressions

With the Debugger, all expressions are *integer* expressions.
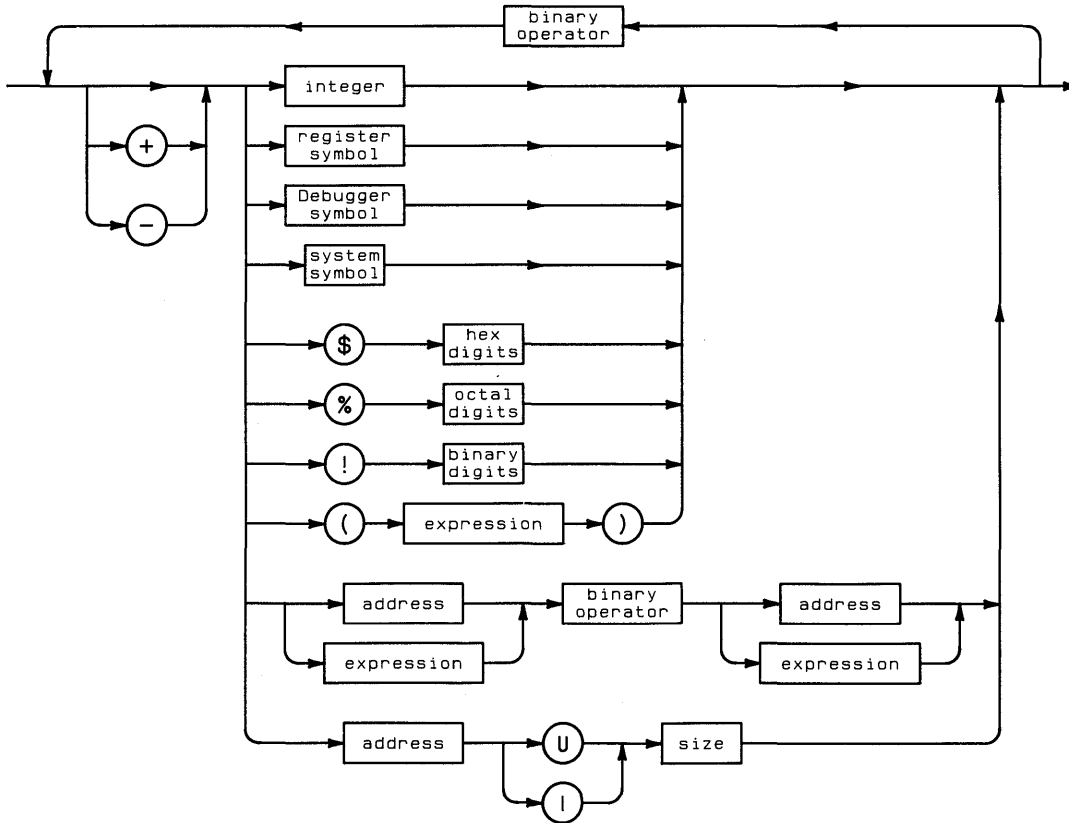
Item	Description	Range
binary operator	an operator that requires two operands	$+$, $-$, $/$, $*$, $<$, $<=$, $=$, $>=$, $>$, $<>$
register symbol	a symbol representing a processor register	A0..A7, D0..D7, PC, SP, US, SR
Debugger symbol	a symbol known to the Debugger	LN (Line Number)
		EC (Escape Code)
		IO (I/O result code)
		GB (the Global variable Base)
		RB (the code Relocation Base)
		SF (the current Stack Frame pointer)
system symbol	any symbol in the system symbol table	—
address	a numeric expression followed by a "^", which refers to the contents of the specified memory address	-2^{31} thru $2^{31} - 1$
size	integer expression that specifies the number of bytes to be used	1 thru 4

The "U" (unsigned integer) and "I" (signed integer) option paths indicate whether the value at the specified address and with specified number of bytes (size) is to be treated as a signed or unsigned integer.

Multiple Commands on a Line

Several commands may be entered on the same line. These commands are separated by a semicolon (;).
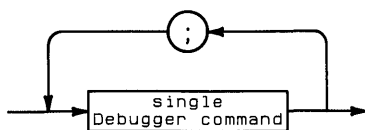
Breakpoint Commands

Breakpoints are points in a program where execution may be halted. The Breakpoint commands control program execution by setting up, activating, and clearing breakpoints in a program.

B

The "B" command causes the breakpoint table to be displayed.
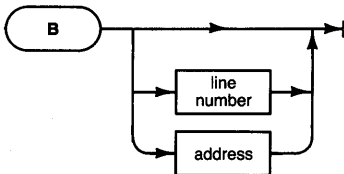
Item	Description	Range
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

The first column contains an "A" for an active breakpoint or a "D" for a deactivated breakpoint. If no location is specified, the table displays all breakpoints.

BA

The "BA" command Activates disabled breakpoints. If a location is specified, then only that breakpoint is re-activated; otherwise, all breakpoints are re-activated.
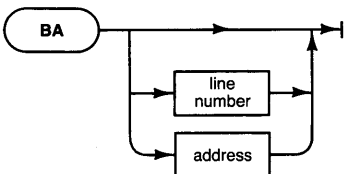
Item	Description	Range
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BC

The "BC" command Clears breakpoints. If a location is specified, then only that breakpoint is cleared; otherwise, all breakpoints are cleared.
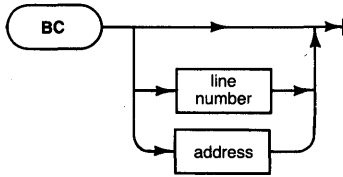
Item	Description	Range
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BD

The "BD" command De-activates breakpoints. If a location is specified, then only that breakpoint is de-activated; otherwise, all breakpoints are de-activated.
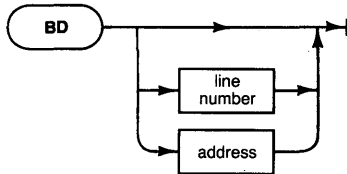
Item	Description	Range
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$

BS

Setting breakpoints with the "BS" command causes the program to stop or perform some operation at a given line number or instruction address.
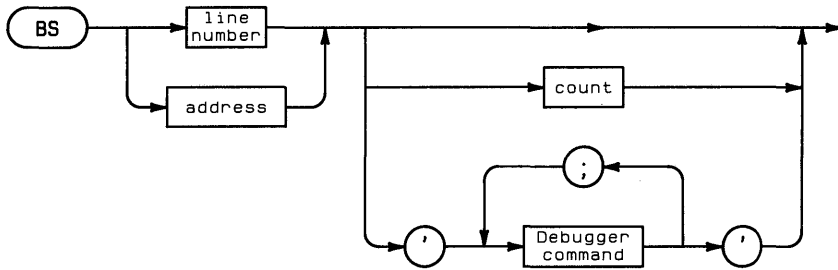
Item	Description	Range
line number	an expression that identifies a program line	0 thru 2^{16}-1
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	expression	-2^{31} thru $2^{31}-1$
Debugger command	any legal commands delimited with single or double quotes	—

If only a location is specified, the breakpoint is set at that location and then activated. The program will halt just *before* it subsequently reaches that point.

Specifying a count sets a breakpoint that will halt the program after the count has been decremented to 0. (The count is decremented each time the location is reached.) When the program is halted, this type of breakpoint is automatically cleared. (The other two types of breakpoints set with the BS command are not cleared when encountered.)

Adding a command string to the breakpoint causes the command to be executed each time the point is reached. A "?" in the command string causes the Debugger to wait for input from the keyboard. Otherwise, the command is executed and program execution resumes.

The Call Command

This command is used to call the subroutine at the specified address.
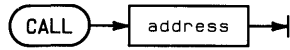
```
(CALL)──▶│ address │──▶│
```

Item	Description	Range
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31} -1$

The effect of this command is as if a Jump to Subroutine (JSR) instruction was encountered just before the current program counter (PC).

The CALL command can be abbreviated with the letters CA.

Display Command

D

The D command is like a print statement where the parameters are objects and formats.
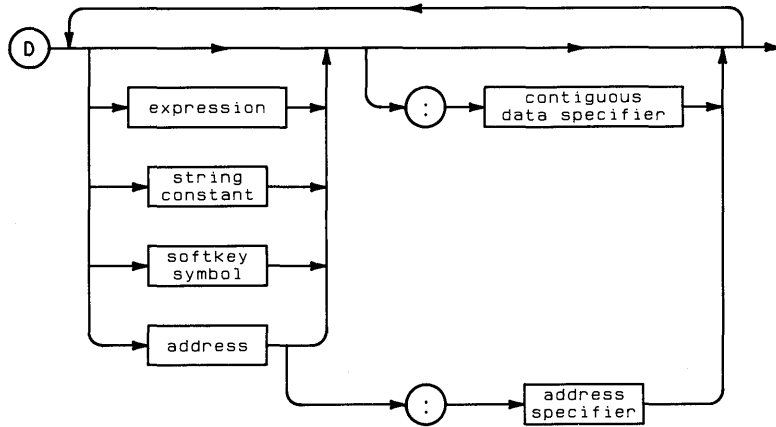
contiguous data specifier:
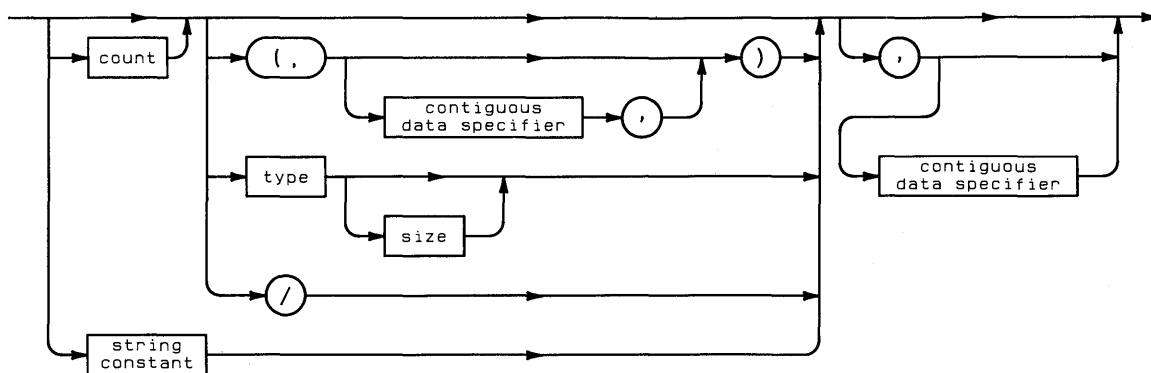
Item	Description	Range
expression	integer expression	-2^{31} thru $2^{31}-1$
string constant	literal value	any character delimited with single or double quotes
softkey symbol	a symbol (not the actual key)	"K0" thru "K9"
address	an expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$

address specifier:
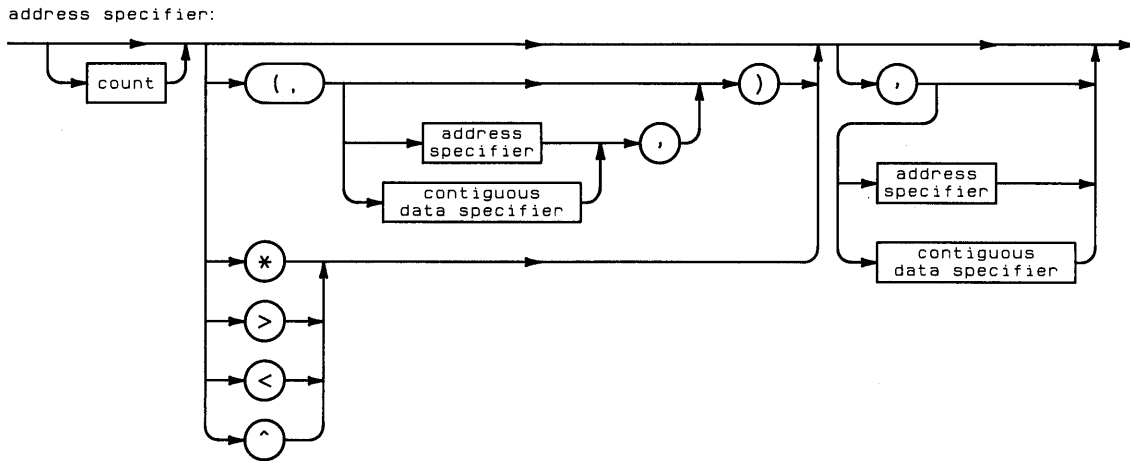
Item	Description	Range
type	A = Alpha character B = Binary H = Hexadecimal I = Integer (size = 1..4) O = Octal (size = 1..4) S = String type (size is declared size) R = Real (size not allowed) U = Unsigned integer X = reverse assembly (size not allowed)	
size	integer constant	-2^{31} thru $2^{31}-1$ (except where noted above)

Objects can be immediate, direct, or indirect.

Formats describe the internal representation of the data. Non-consecutive data can be displayed using the format options available when the address parameter is used.

Dump Commands

These commands allow you to perform the DUMP ALPHA and DUMP GRAPHICS functions while in the Debugger.

DA

The DA command performs the DUMP ALPHA function.

DG

The DG command performs the DUMP GRAPHICS function.
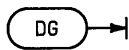
Note

These commands can only be used while executing programs in the processor's "user mode." If attempted while in "supervisor mode," the following error will be reported:

```
NOT ALLOWED NOW
```

Escape Code Commands

These commands allow you generate and trap escape codes while in the Debugger.

EC

The effect of executing this command is the same as if you had executed an ESCAPE(code) in the program just before the current PC. If any ET, ETC, or ETN commands have been used to set up escape code trapping, then the Debugger will be halted and the escape code displayed on the screen.
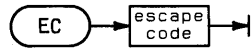
Item	Description	Range
escape code	signed integer expression; negative for system escapes, positive for user escapes.	-2^{15} thru 2^{15}

Here is an example display:

```
>EC 10
-EXCEPTION-
ESCAPE CODE        +10
SR=$0004 PC=    -228230 LINE            +9
```

ET

The Escape Trap command allows you to specify that either all escape codes or specified escape codes are to be trapped by the Debugger.
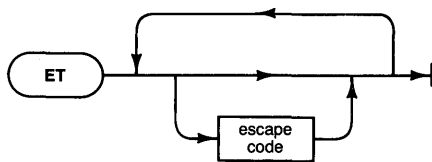
Item	Description	Range
escape code	signed integer expression; negative for system escapes, positive for user escapes	-2^{15} thru $2^{15}-1$

If an escape code that is in the list is encountered, execution stops and control is given to the Debugger. If no escape codes are specified, then processing stops for all escape codes.

Up to 4 escape codes may be specified with the ET command.

ETC

The Escape Trap Command allows you to set up command(s) to be executed when an ESCAPE is generated.
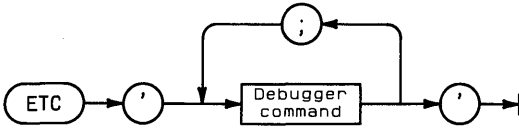
Item	Description	Range
Debugger command	command to be executed when an escape is encountered	any valid Debugger command

ETN

The Escape Trap Not command specifies that processing should stop for all escape codes *except* the ones listed. If none are listed, then processing won't stop for *any* escape codes.
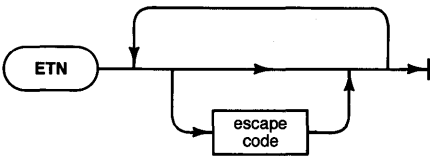
Item	Description	Range
escape code	signed integer expression; negative for system escapes, positive for user escapes	-2^{15} thru $2^{15}-1$

If the program was started with the D command, then ETN -20 0 (which traps all except the [Stop] key and normal program termination) is in effect.

Up to 4 escape codes may be specified with the ETN command.

Format Commands

The format commands allow you to specify the default display format.

FB

The Format Binary command sets the default format to Binary values.

```
( FB )—◄
```

FH

The Format Hex command sets the default format to Hexadecimal values.

```
( FH )—◄
```
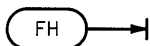
FI

The Format Integer command sets the default format to signed Integer values.

```
( FI )—◄
```
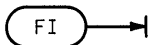
FO

The Format Octal command sets the default format to signed Octal values.

```
( FO )—◄
```
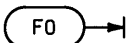
FU

The Format Integer command sets the default format to Unsigned integer values.

```
( FU )—◄
```

Go Commands

The Go commands control program execution by telling the Debugger how many lines to execute or the line at which to halt.

G

The "G" command causes normal execution to resume. If a count option is used, that number of statements is executed.
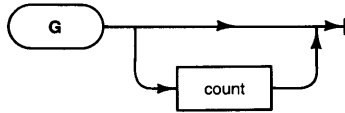
Item	Description	Range
count	integer expression	1 thru $2^{15}-1$

GF

The "GF" command is the same as the "G" command except execution is slowed and line numbers are Flashed in the lower right corner of the CRT.
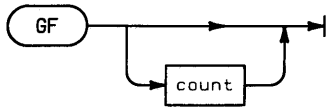
Item	Description	Range
count	integer expression	1 thru $2^{15}-1$

GT

The "GT" command causes execution to Go 'Til the specified location is reached.

Item	Description	Range
line number	an expression that identifies a program line	0 thru $2^{16}-1$
address	integer expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$
Debugger command	any legal Debugger commands delimited with single or double quotes	

If a count option is used, execution continues until the location is reached that number of times.

If the Debugger command option is used, the command(s) are executed when the location is reached.

GTF

The "GTF" command is the same as the "GT" command except execution is slowed and line numbers are flashed in the lower right corner of the CRT.

Item	Description	Range
line number	an expression that identifies a program line	0 thru $2^{16}-1$
address	integer expression, followed by a "^", that identifies a location in memory	-2^{31} thru $2^{31}-1$
count	integer constant	1 thru $2^{31}-1$
Debugger command	any legal Debugger commands delimited with single or double quotes	

IF, ELSE, and END Commands

These commands allow conditional execution of Debugger commands.
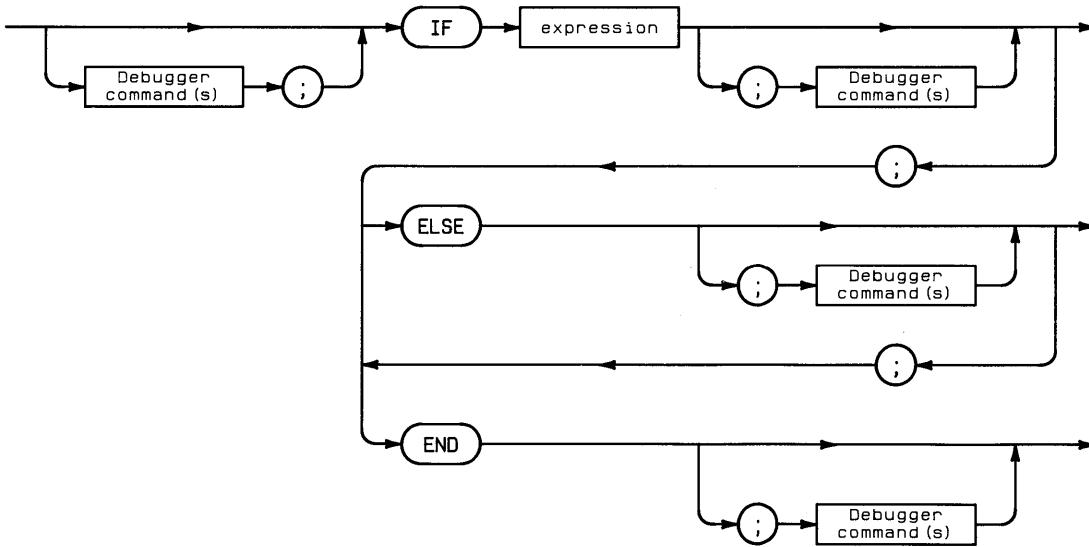
Debugger command(s):
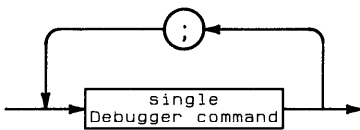
Item	Description	Range
Debugger command(s)	command(s) to be executed when the sense bit is TRUE	see drawing
expression	numeric or boolean expression whose value determines the state of the "sense bit"	any valid Debugger expression
single Debugger command	one Debugger command	any valid Debugger command described in this reference section

In order to better understand how IF, ELSE, and END statements work, you need some background information. There is a sense bit that determines whether or not Debugger commands are executed. This sense bit is set to TRUE at the beginning of every command line. Commands on the line are executed as long as this bit is TRUE and skipped when the sense bit is FALSE.

When an IF statement is encountered, the expression is evaluated. If it evaluates to non-zero or TRUE, then the sense bit is set TRUE. Subsequent commands are executed while this bit is TRUE. When an ELSE command is encountered, the sense bit is complemented (i.e., if it was TRUE, then it is set to FALSE, and vice versa). When an END statement is encountered, the sense bit is set to TRUE. Here is an example of this situation:

```
>IF 1=1;D 'NON-ZERO';D 'TRUE';ELSE;D 'ZERO';D 'FALSE';END;D 'ALWAYS'
NON-ZERO
TRUE
ALWAYS
```

Here is an example of the converse situation.

```
>IF 0;D 'NON-ZERO';D 'TRUE';ELSE;D 'ZERO';D 'FALSE';END;D 'ALWAYS'
ZERO
FALSE
ALWAYS
```

Notice that the commands after the END statement are always executed. Note that the IF statement does not have to be the first command in the line.

The ELSE command can be abbreviated as EL; the END command can be abbreviated as EN.

Open Memory Commands

These commands allow you to examine, and optionally modify, the contents of memory locations.

OL, OW, OB

The Open Byte, Open Long, and Open Word commands are used to examine consecutive memory locations and to assign values to the locations.
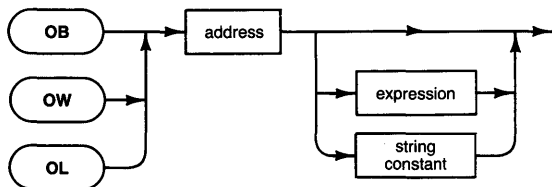
Item	Description	Range
address	an expression, followed by a "^", that identifies a location in memory (with OW and OL, the address must be an even number)	-2^{31} thru $2^{31}-1$
expression	integer expression	-2^{31} thru $2^{31}-1$
string constant	literal	any character delimited with single or double quotes

Semantics

When no value is specified after the location, the location and the contents of the location are displayed and followed by a special prompt. The prompt is for an up-arrow key, a down-arrow key, or the [Return] or [Enter] key, or a numeric expression followed by the [Return] or [Enter] key.

- The up-arrow key causes the next higher location and value to be displayed and the special "Open" prompt.

- The down-arrow key is the same except that the next lower address is displayed.

- The [Return] or [Enter] key causes termination of the "Open" prompt and a return to the standard Debugger prompt.

- A numeric followed by [Return] or [Enter] will place the value of the expression into the current location.

The amount of the increment/decrement is as follows:

- 1 byte for the "OB" command
- 2 bytes for the "OW" command
- 4 bytes for the "OL" command

When the Open memory commands are invoked with value options, the specified value is assigned to the corresponding location. No attempt is made to read the corresponding memory location.

Procedure Commands

These commands allow you to halt the program when a procedure is called or exited. Both of these commands will only work if the procedures were compiled with $DEBUG ON$.

PN

The PN (Procedure Next) command halts the Debugger when a procedure is called by the current procedure or main program (or when the current procedure is exited).

When the current procedure or main program calls another procedure, the Debugger displays NEXT PROC and halts the program *before* executing the first line of the called procedure.

If the current procedure is exited before another is called, the message PROC EXITED is displayed and the Debugger is halted *before* executing the first line of the procedure that called the current one.

PX

The PX (Procedure eXit) command allows you to halt program execution when the current procedure is exited.
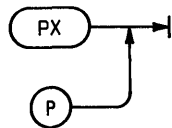
When the current procedure is exited, the message PROC EXITED is displayed and the program is halted *before* executing the next line of the procedure that called the current one. Calling a procedure while in the current one is not reported (as with PN).

Queue Commands

The Queue commands control and display the Queue, which is a record of the line numbers of statements (or memory addresses of processor instructions) encountered during the execution of a program. Note that the program being debugged must have been compiled with $DEBUG ON$ for line numbers to enter the Queue.

Q

The "Q" command displays the addresses or line numbers and addresses of the most recent statements executed since a "QS" command or the start of execution of the current program.

(Q)————
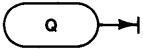
"MORE" is given as a prompt when part of the Queue has been displayed and there is more to come; a reply of ⎡Return⎤, ⎡Enter⎤, or ⎡Select⎤ (**EXECUTE**) will cause the next 1..21 Queue entries to be displayed. Any other reply will be interpreted as another command.

QE

"QE" ends the recording of information in the Queue

(QE)————

QS

"QS" starts the recording of information in the Queue

(QS)————

Register Operations

With the Debugger, you can display or alter the contents of processor registers.
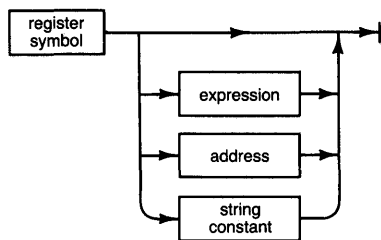
Item	Description	Range
register symbol	A0...A7, D0...D7, SP, US, SR, PC	-
	AA = All Address registers	
	DD = All Data registers	
expression	integer numeric expression	-2^{31} thru $2^{31}-1$
string constant	any character delimited with	-

If a value follows the register symbol, that value is assigned to the register. Otherwise, the current value of the register is displayed. Without the assignment, the command is the same as the D command.

"AA" and "DD" cannot be used to assign values.

Softkey Commands

The Softkey commands allow you to define System softkeys to display literal values and commands, so that these keys will be used as typing aids.

"K0" thru "K9"

These commands allow you to define the softkeys as typing-aid keys; when the softkey is subsequently pressed, it puts the string constant or the result of evaluating the integer expression into the Debugger command-input line.
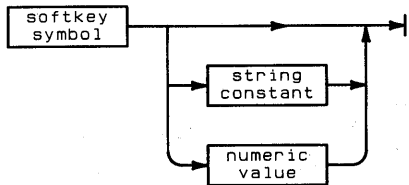
Item	Description	Range
softkey symbol	literal symbol that denotes a softkey	K0 thru K9
string constant	literal value	any characters delimited with single or double quotes
numeric value	integer expression	-2^{31} thru $2^{31}-1$

Numeric values or command strings can be assigned to the softkey symbols K0 through K9 by typing the *softkey symbol* (not by pressing the actual key) and then typing the value to be assigned to the key.

If a string constant is assigned to the softkey symbol, subsequently pressing the corresponding softkey will cause the literal value to be placed in the Debugger command-input line. If a numeric expression is assigned to the softkey symbol, the *result* of evaluating the expression is placed in the input line.

After the command on a line is completed, pressing ⌜Return⌝, ⌜ENTER⌝, or ⌜Select⌝ (⌜EXECUTE⌝) causes the line to be interpreted.

To see the string constant or numeric value that is currently assigned to a softkey symbol, type the *softkey symbol* and press ⌜Return⌝.

System Boot

This command causes the computer to boot the system defined by the parameters given.
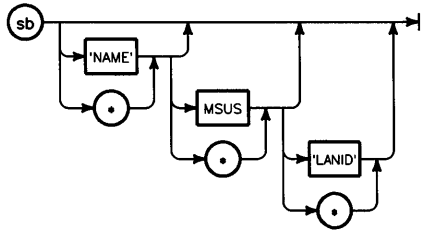
Item	Description	Range
NAME	String or string constant	1 to 10 characters (Name of boot file)
MSUS	Mass Storage Unit Specifier	file system unit number or 8 hex digit value
LANID	String or string constant	12 digits (boot system across the LAN)

Note that sb must be entered in lowercase letters (use the [Shift] key to generate lowercase letters in the Debugger).

Trace Commands

These commands are primarily for use with Assembly language debugging. Note that these commands ignore address breakpoints.

T

The Trace command with count specification causes that number of machine instructions to be executed. A TD command is executed after each machine instruction.
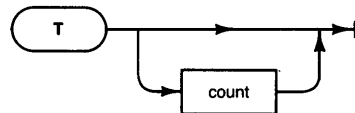
Item	Description	Range
count	integer constant	1 thru $2^{31}-1$

TD

The TD command displays the command string which is defined by the ⬚f4 (⬚k4) softkey.
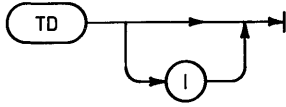
At power up, ⬚f4 is defined to display the PC, the instruction at (PC), the status register, the SP, and all the A (address) and D (data) registers. This display may be altered by changing the definition of ⬚f4.

The optional parameter "I" restores the initial definition to softkey ⬚f4.

TQ

Causes execution of machine instructions until the specified address is reached. Then a "TD" is executed. This command also records the Program Counter (not line number) values in the Queue.
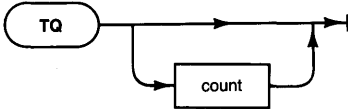
Item	Description	Range
count	integer constant	1 thru $2^{31}-1$

TT

Same as the "GT" command except that PC values are recorded in the Queue and a "TD" is executed after the last machine instruction is executed.
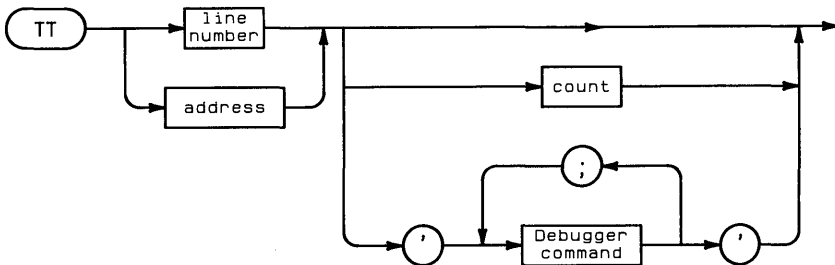
Item	Description	Range
line number	an integer numeric expression identifying a program line.	0 thru $2^{16}-1$
address	an integer numeric expression followed by a "^"	$-2^{31}-1$
count	integer constant	-1 thru $2^{31}-1$
Debugger command	Any legal commands delimited with single or double quotes	-

Walking the Procedure Links

WD

The Walk Dynamic link command causes execution of an "SF" command and then the Debugger symbol, "SF", takes the value of the dynamic link from the current Stack Frame. Another "SF" command is then executed.

```
( WD )——►|
```

WR

The Walk Reset command restores A6 to "SF".

```
( WR )——►|
```

WS

The Walk Static link command is the same as the "WD" command except that the Debugger symbol, "SF" takes the value of the static link. This brings you to the Stack Frame of the nesting procedure as opposed to the calling procedure. Level 1 procedures have no static link.

```
( WS )——►|
```

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.

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.

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.

Index

a

b

C

d

e

i

m

n

o

q

r

W

Z

Please use this Reader Comment Card to evaluate this document and tell us of problems or suggest improvements. **SERIOUS ERRORS** rendering a product or device inoperative should be entered in STARS (Software Tracking and Reporting System) by the HP Response Center or your Support Engineer.

Please rate the quality of each item below in terms of your expectations:

	Far Below Expectations	Below Expectations	Meets Expectations	Exceeds Expectations	Far Exceeds Expectations
Retrievability:	1	2	3	4	5
Manual Title:	1	2	3	4	5
Table of Contents:	1	2	3	4	5
Tabs:	1	2	3	4	5
Headings in Chapters:	1	2	3	4	5
Cross-References:	1	2	3	4	5
Task References:	1	2	3	4	5
Index:	1	2	3	4	5
Organization:	1	2	3	4	5
Completeness:	1	2	3	4	5
Accuracy:	1	2	3	4	5
Readability:	1	2	3	4	5
Language Usage:	1	2	3	4	5
Layout:	1	2	3	4	5

Recommended improvements (attach additional information if needed):

_____

_____

_____

_____

_____

Name: _____   Company: _____

Job Title: _____   Address: _____

Phone: _____   _____

Please enter the series number of your HP 9000 system, e.g. 200 or 300: _____

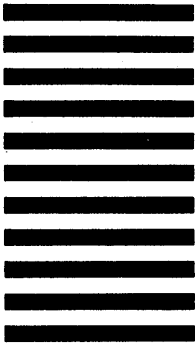Hewlett-Packard has the right to use submitted suggestions without obligation, with all such ideas becoming property of Hewlett-Packard.