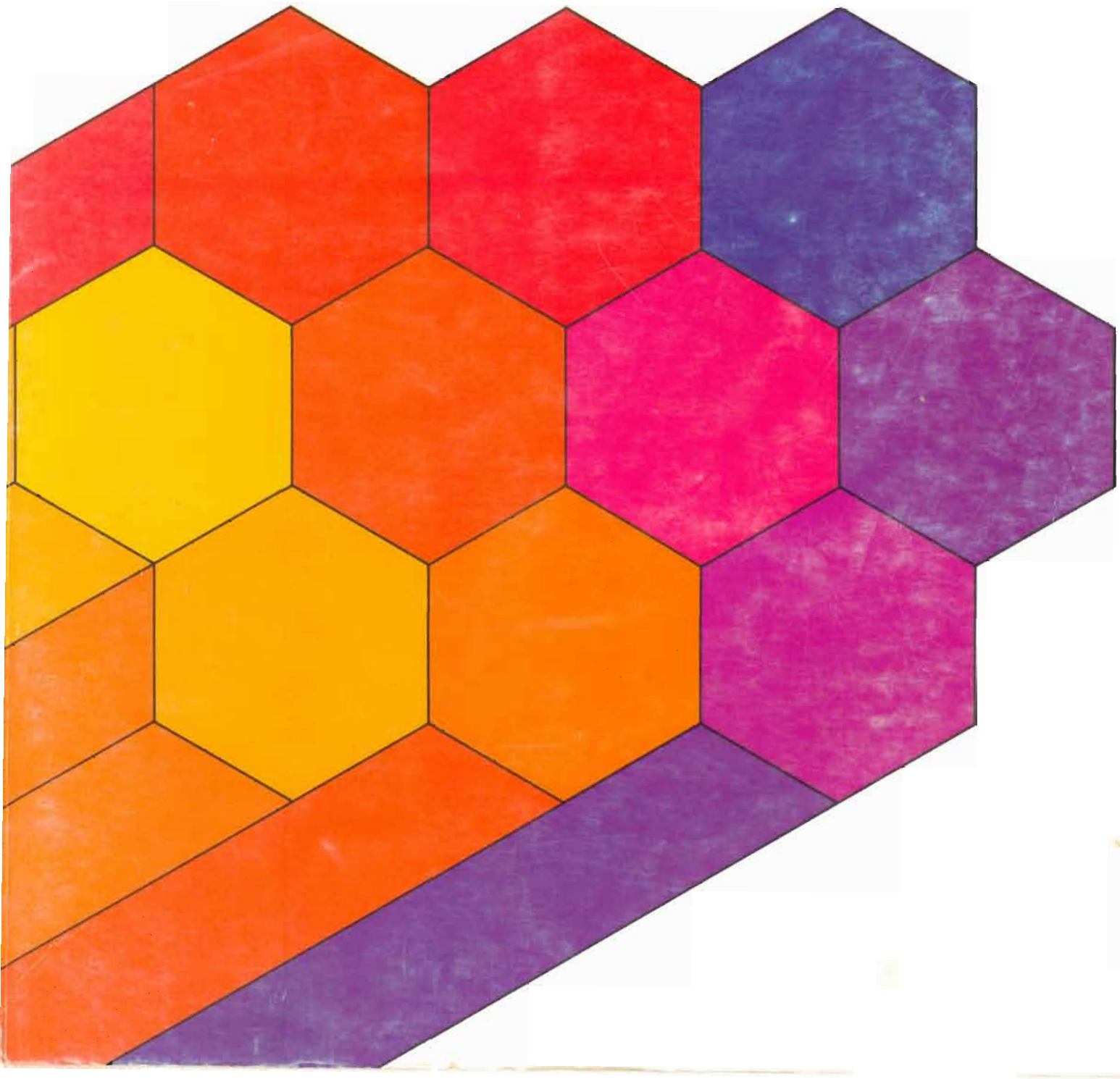
HEWLETT-PACKARD

UCSD p-System™ Version IV.1 Operating System Reference Manual



HEWLETT PACKARD

UCSD p-System Version IV.1

Operating System Reference Manual

December 1982

Reorder Number
00087-90381

Printed in U.S.A.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

TABLE OF CONTENTS

VI THE ADAPTABLE ASSEMBLER

VIII CONCURRENT PROCESSES

IX UTILITIES

I. INTRODUCTION

I.1 How to use this Manual

This is the basic reference to the UCSD p-System. It should contain answers to your questions concerning the System once it is up and running, but it is not meant to be a tutorial about the use of the System. If you have never used the UCSD p-System before, you should consult Introduction to the UCSD p-System.

Although this manual is not tutorial, this introduction is designed as a description of the overall structure of the System, and you should read it before doing any extensive work. Once you have read the introduction manual and gained some experience, especially a feel for our Text Editor and file-handling, then you should approach this manual. Read the remainder of this chapter, the following chapter on System commands, and whichever discussions are most appropriate to your work -- whether the Adaptable Assembler or less general-purpose matters. The chapters on the Filer and the Screen Oriented Editor will also prove useful. If you intend to work with large programs, you should definitely read chapter VII: Segments, Units, and Linking.

Much of the manual -- e.g., sections on utilities, YALOE, and the appendices -- is best used as a reference when you need some specific information, such as the use of a utility, the meaning of a particular error message, and so forth.

Other high-level languages, such as Pascal and FORTRAN, are described in manuals of their own.

It is best to start slowly. If you do that, your progress will in fact be rapid; it is most confusing to try to do too much at once. The System is designed for easy use, and you will find that most tasks can be accomplished with relatively few simple commands.

In any case, we believe that the best way to learn our System is to temper use of all this documentation with liberal use of the software while it is running -- whether you begin with rudimentary but useful programs, or out-and-out play. This is the only way to develop a personal feel for our environment, which will allow you to develop your own ways of using the System. In time you will learn to use subtler forms of the commands, and develop your own shortcuts.

We hope that you are creative in your use of our System, since such work is capable of benefitting all of us, and since enjoyment and productivity go hand in hand.

NOTE: we have tried to keep this manual free from arcane conventions. A couple of things seem worth mentioning, however. Angle brackets ('<' and '>') are used throughout in their common sense of indicating a meta-object or the generic name of something; thus, single keystrokes with long names are represented this way (<return> or <escape>), and names of things within a literal description are represented this way as well:

IF <Boolean expression> THEN <statement> ELSE <statement>

... and so forth. Also, ranges of numbers are shown as in Pascal syntax, with a two-dot (rather than three-dot) ellipsis, for example:

0..9 -32768..32767 5..70 -1.999..+1.999

I.2 Overview

This section discusses the general organization of the System, its file and device structures, and general mechanisms for organizing programs. This is not an introduction to using the System, but it should give you a perspective on the System's aims and rationales.

I.2.1 System Rationale and Organization

The UCSD System was initially designed as a program development system for microcomputers. Originally it was used to teach programming, but was soon put to a variety of uses, including its own development. The current system contains many functions and capabilities that were not present in the original, but many of the original design assumptions still remain: a simple, menu-driven operating system, a relatively small main memory, an interactive terminal (typically, a CRT), and relatively low-capacity on-line storage media (typically, a floppy disk or two).

The simplest of machines running the UCSD p-System will have a box with the central processor, a CRT (a line-oriented terminal will serve, but is inconvenient), and a floppy disk drive. The floppy drive will contain a disk with the system software, and some number of user files. The user files might be source programs, object programs, raw data, or document text.

This simple system can be and usually is extended with such things as more or better disk drives, a printer, and other devices. All the same, it will be used by a single user sitting at the CRT and either developing programs or running existing ones. The System is geared toward this interactive environment.

The Operating System, Filer, and Editor are all "menu-driven" -- a promptline is continually displayed at the top of the screen with all (or nearly all) of the current commands visible. These commands are invoked by a single keystroke, and the organization is hierarchical. That is, typing a key generally causes either an action to be performed, or another promptline to be displayed which details new commands at a different and "lower" level.

This document, particularly Chapter II, talks about the "command level". That is the highest level of the Operating System, and the one visible to the user as the promptline which appears when the system is first booted. The commands at this level are straightforward and self-explanatory: R(un, C(ompile, E(dit, F(ile, and so forth. Some of them, such as R(un and C(ompile, cause actions to be performed directly on a file. Others, such as E(dit and F(ile, invoke those particular programs, which are themselves menu-driven.

The Filer and Linker and certain utilities perform functions traditionally performed by larger operating systems. In the UCSD System they are treated as separate

programs (just as user programs are), and so are not properly part of the Operating System. Below the "outer" or "highest" command level, the Operating System is not visible to the user, but remains an important component of the System by being available for continual monitoring and control of running programs and I/O devices.

When first bringing up the System, the user may have to contend with some low-level details, depending on the hardware involved. This aspect of the System is outside the scope of this manual, and if you are interested in it, you should consult the Installation Guide.

Bringing up the System starts the System's Interpreter (also called the Emulator) which executes all programs compiled from a high-level language. While the Assembler generates machine language for the HP-86/87 processor, the System's high-level languages Pascal and FORTRAN are compiled to an intermediate language called P-code. This code is in the form of machine code for an idealized "P-machine". To execute it on an actual processor, the P-machine must be emulated: either by an interpreter that processes operations at runtime, by a code generator that performs the translation prior to runtime, or by a hardware implementation that executes P-code directly. All of these methods are in use, but most current installations use an interpreter; this is the method first employed at UCSD. More information on the P-machine can be found in the Internal Architecture Guide.

This introduction has now enumerated all of the components of the System, albeit in a very cursory way. The following two illustrations should clarify the relation between the Operating System and the remaining System components. Figure 1 shows a tree which represents the command structure: typing various commands within the System amounts to a traversal of this tree. Figure 2 is a more detailed picture of various major components and their interrelationships.

For more detailed information on the various System commands, refer to Chapter II.

All components of the System exist as files which are usually stored on floppy disks. The same is true for all user-generated software. The logical next step, then, is to examine the System's treatment and organization of files.

Command tree — only the most frequently used commands are shown

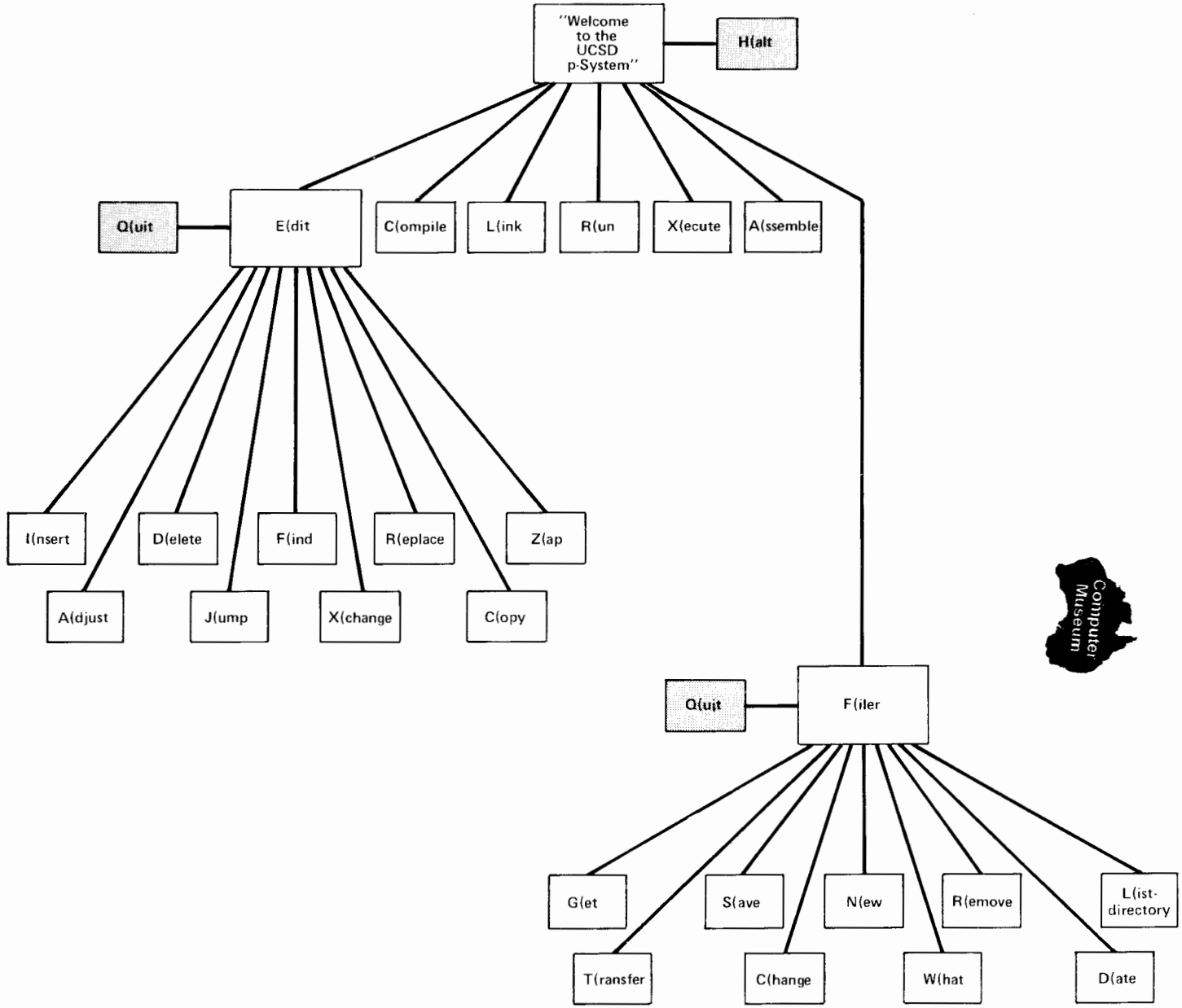


FIGURE 1

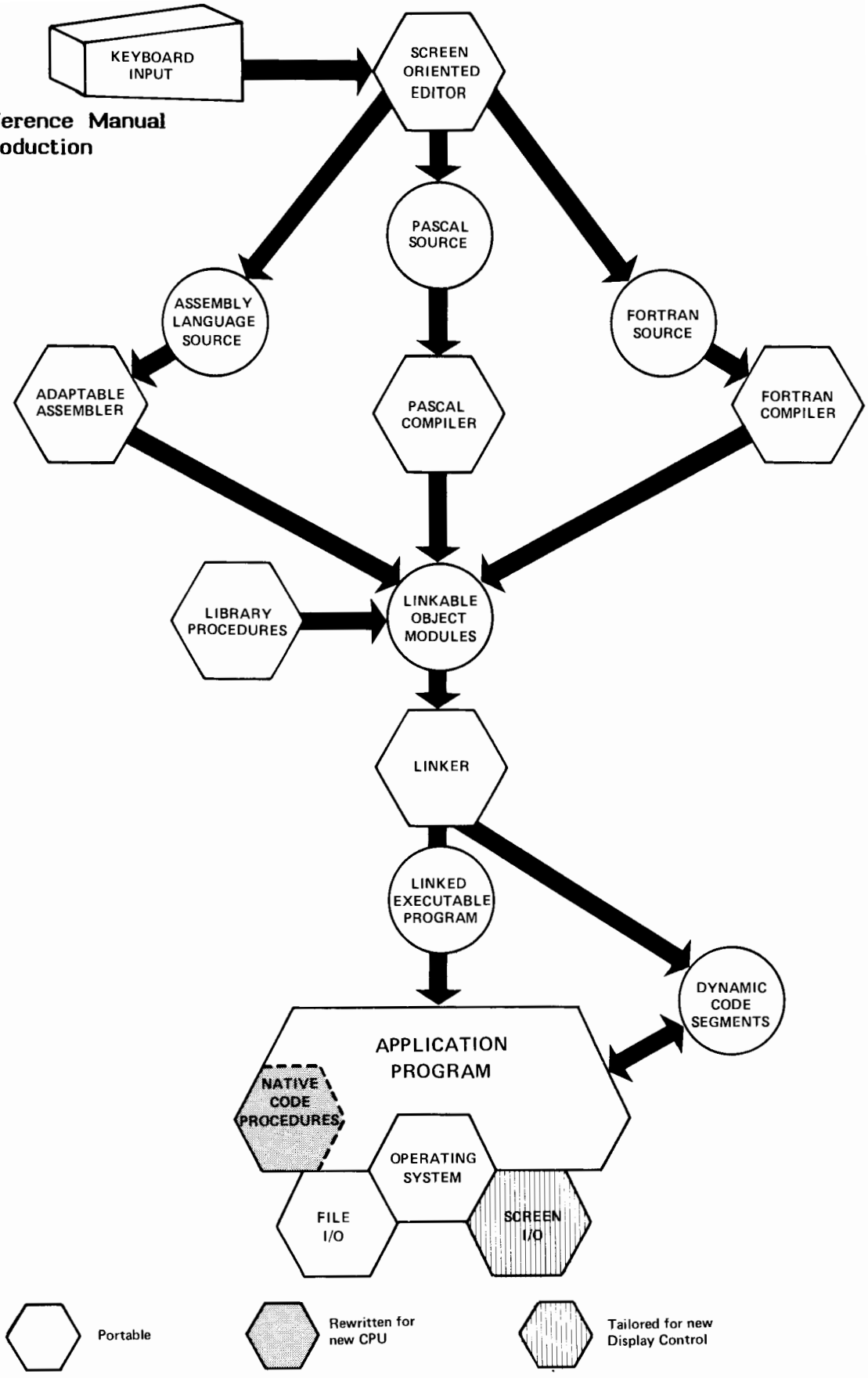

KEYBOARD INPUT
SCREEN ORIENTED EDITOR
PASCAL SOURCE
ASSEMBLY LANGUAGE SOURCE
FORTRAN SOURCE
ADAPTABLE ASSEMBLER
PASCAL COMPILER
FORTRAN COMPILER
LIBRARY PROCEDURES
LINKABLE OBJECT MODULES
LINKER
LINKED EXECUTABLE PROGRAM
DYNAMIC CODE SEGMENTS
APPLICATION PROGRAM
NATIVE CODE PROCEDURES
OPERATING SYSTEM
FILE I/O
SCREEN I/O
Portable
Rewritten for new CPU
Tailored for new Display Control

FIGURE 2

1.2.2 File Organization

A file, to the UCSD System, is a collection of data. It may reside on a disk, and be brought into main memory only when it is being directly used by the System or a user program. It may be data that a program reads from a peripheral device, or sends to a peripheral device.

A file may contain any sort of data and be organized in any way, but the System will treat certain files in very specific ways, and there are naming conventions which support this special treatment. The naming conventions inform the System how to treat a given file, and also serve as mnemonics for the user.

Before discussing the individual file types, it should be mentioned that "disk files" are stored on some random access medium, usually a floppy disk. Each such disk contains a directory which describes up to 77 files. File size varies, and the limits depend upon particular hardware (more information along these lines is given in the Installation Guide).

File manipulation is usually done with the Filer. The Filer is a program which is invoked at the outer command level. It provides a variety of commands which allow for the creation, naming, and renaming of files, their removal, and their transfer between different devices (disk drives, printers, CRTs, and the like). It also provides for some management of storage units themselves. More information on this is provided below in Section 1.2.3, and the Filer is described thoroughly in Chapter III.

Note: bootstrapping the System involves reading files off of a particular disk. That disk is called the "System disk." In the System's syntax for filenames, it is called '*', and when a disk name is shown preceded by a star (e.g., *SYSTEM.PASCAL), that means the file is on the system disk. This convention is used throughout this manual. More information on device names and filenames appears in chapter III.

1.2.2.1 System Files

The files which comprise the major portions of the System itself are identified by the prefix 'SYSTEM.'. Thus, important files are SYSTEM.PASCAL, SYSTEM.EDITOR, SYSTEM.ASSMBLER, and so forth. Which files are actually shipped with a given system, and on which disks, is discussed in the Introduction to the UCSD p-System. This section gives a general description of the names of the major pieces of the System.

The Operating System itself is SYSTEM.PASCAL. Some of its major pieces are:

SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.LINKER
SYSTEM.COMPILER
SYSTEM.ASSMBLER (note the missing 'E')

... all of these are programs are directly called by single-letter commands at the outer command level.

SYSTEM.SYNTAX

... contains all the Compiler's error messages.

SYSTEM.COMPILER is not necessarily Pascal -- it could contain any of the available compilers (currently, Pascal or FORTRAN). In this way, by changing file names a user may change the compiler that is accessed by one keystroke.

Similarly, SYSTEM.EDITOR is shipped as the Screen Oriented Editor, and usually contains that code. But should you be constrained to using a line-oriented terminal, you might change YALOE (Yet Another Line Oriented Editor) to SYSTEM.EDITOR, because it would better suit your needs.

SYSTEM.LIBRARY

... contains previously compiled or assembled routines to be linked in with other programs.

SYSTEM.STARTUP

... is an executable codefile. If a file with this name exists when the System is bootstrapped or I(nitialize'd, it is executed before the main System promptline is displayed. This is aimed at providing a turnkey environment for users who desire one.

SYSTEM.MISCINFO

... is a data file containing miscellaneous data items about an individual system -- most of it is devoted to terminal-handling information.

SYSTEM.INTERP

... is the system emulator.

There are three other SYSTEM. files that are commonly, though not always, present. These are the files that make up the user's workfile, and since they are handled in a special way, and relate directly to individual use of the System, they are discussed separately in Section 1.2.2.3 below. Before discussing workfiles, we will talk about more ordinary user files.

When the System is bootstrapped, certain System files must be on the disk it is bootstrapped from. Other system files may be anywhere. The System will search for them whenever it is bootstrapped or I(nitialized (see Chapter II), and whenever it needs them and they are not on the device where it previously found them. First the System will search the System disk, and then any other disks that are on-line. A description of what files must be present on a disk that bootstraps is found in the Installation Guide, Chapter IV.

1.2.2.2 User Files

User files are generally one of three things: program or document text, compiled or assembled program code, or other data in any sort of user-defined format. Some naming conventions cover these files as well, and in particular, correspond to these three types -- the suffix of a filename indicates which type of file it is.

.TEXT files, such as SORTER.TEXT, NONSENSE.TEXT, or even SYSTEM.WRK.TEXT, are human-readable files, formatted for use by the System's Editor (typically the Screen Oriented Editor). They include a header block, and follow certain internal conventions.

.CODE files, such as SORTER.CODE, FISBIN.1.CODE, or SYSTEM.WRK.CODE, are either P-code or 'native code'. P-code is the code generated by the System's compilers and executed on the P-machine. Native code refers to code that is

ready to run on some particular processor. .CODE files are typically the output of a compiler or an assembler; they may also be generated by the Linker from a group of previously existing codefiles.

.DATA files such as FOR.SORT.DATA contain information for user programs, in some format known to the user.

These naming conventions in general do not matter to the Filer; Filer commands refer to any file regardless of name. The exceptions to this are the G(et, S(ave, and N(ew commands, which deal with the workfile -- these are described below.

These naming conventions do matter to certain other System programs -- for example, the Editor will only edit .TEXT files. A codefile must be created with the .CODE suffix; once it is created, the name can be changed to something else, and it will still be eX(ecute'able. The compilers and assemblers automatically append .CODE to the names of output files you specify. This Manual describes these and other such conventions wherever they are relevant.

Another suffix you may encounter is .BAD, for the immobile files used to cover physically damaged portions of a disk.

More details about file formats are given at the beginning of Chapter III.

1.2.2.3 The Workfile

The user may designate a 'workfile', which can be thought of as a scratchpad area for keeping new and unnamed material. Many System programs assume you are working on the workfile unless you specify otherwise. The workfile may be created by designating existing files, or by creating a new file with the Editor.

Modifying the workfile can cause temporary copies to be generated, which (until they are saved) are named:

SYSTEM.WRK.TEXT
SYSTEM.WRK.CODE and
SYSTEM.LST.TEXT

SYSTEM.WRK.TEXT can be created upon leaving the Editor; if it happens to contain a program, then a successful C(ompile or R(un will create SYSTEM.WRK.CODE. If the compilation is successful, the R(un command goes on to execute the code immediately. SYSTEM.LST.TEXT may optionally be created by the Compiler.

Whenever a program contained in SYSTEM.WRK.TEXT is altered by the Editor, R(un will recompile it in order to keep SYSTEM.WRK.CODE up to date.

The Filer can S(ave these files under permanent names. The Filer is also used to designate a new workfile with the G(et command, or remove an old one with N(ew.

1.2.3 Device and Volume Organization

The various peripherals that the System may use are referred to as "devices". When this document refers to a "volume", it means the "contents" of a device. A single disk drive (a device) may be the home for several floppy disks (volumes).

The System distinguishes between block-structured and non-structured devices. Block-structured devices are usually disks. They contain volumes which each contain a directory and various files. Internally a volume is organized into randomly accessible fixed-size areas of storage called "blocks"; a block is 512 bytes. Files may be of variable size, but are always allocated an integral number of blocks. Non-block-structured devices include printers and keyboards and remote lines. They have no internal structure, and deal with serial character streams. Non-block-structured devices may perform input, output, or both; the physical interface to them may be either serial or parallel.

A device or a file may be either a source of data or a sink for data. Many of the Filer's data transfer operations apply to devices as well as to files.

The System and its intrinsics refer to devices by both name and number. Standard devices have standard names, and removable volumes like floppy disks have their names recorded on them. Names and numbers are usually interchangeable. Device names are followed by a ':' (e.g., PRINTER:) to distinguish them from file names, and so they can be prefixed to filenames (e.g., SYSTEM:SAVEME.TEXT).

The name of a device that contains removable volumes (such as a floppy drive) is the name of the volume it contains at any given time. The number of that device never changes.

The name of a disk file includes (as a prefix) the disk it resides on. The System always has one default prefix (when the System is booted it is '*', the System disk) so that the user need not type out the prefix every time a file is needed.

For example, SYSTEM:SAVEME.TEXT and TABLES:SAVEME.TEXT name two different files on two different disks (both files are called SAVEME). These might also be specified as #4:SAVEME.TEXT and #5:SAVEME.TEXT. If the default prefix had been changed by the user to TABLES:, then typing SAVEME.TEXT would be understood to mean TABLES:SAVEME.TEXT.

Here is the complete list of predefined device numbers and names:

Device Number	Volume Name	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTERM:	screen and keyboard without echo
4	<disk name>:	the system disk
5	<disk name>:	the alternate disk
6	PRINTER:	a line printer
7	REMIN:	a serial input line
8	REMOUT:	a serial output line
9..22	<disk name>:	additional disk drives
23	RAMDISC:	electronic disk (system RAM in excess of 160K bytes)

This table is given, with some further exposition, in Chapter III on the Filer. Note that REMIN: and REMOUT: often refer to the same device (for example, a phone line with a MODEM).

This summarizes the System's treatment of devices. Most use of the System does not require more hardware knowledge than that outlined here. From time to time, however, it may be necessary to do some direct device control, some modification of device characteristics, or some messy on-disk file manipulation (such as rescuing partially bad files).

The System accomplishes device control through a portion of the emulator. On most implementations this is called the BIOS (for Basic I/O Subsystem). The BIOS contains device drivers, and a subset of it, called the SBIOS (you guessed it: Simplified BIOS) is modifiable by users who have an Adaptable System. Methods and suggestions for modifying the SBIOS are contained in the Installation Guide. Still more detailed information may be found in the Internal Architecture Guide.

Also described in the Installation Guide are ways to control the System terminal (CONSOLE:). The System's knowledge of CONSOLE: comes from a file named SYSTEM.MISCINFO and a procedure within the Operating System called GOTOXY. SYSTEM.MISCINFO can be modified using a utility program called SETUP, and GOTOXY can be rewritten and bound into the Operating System using the utility LIBRARY.

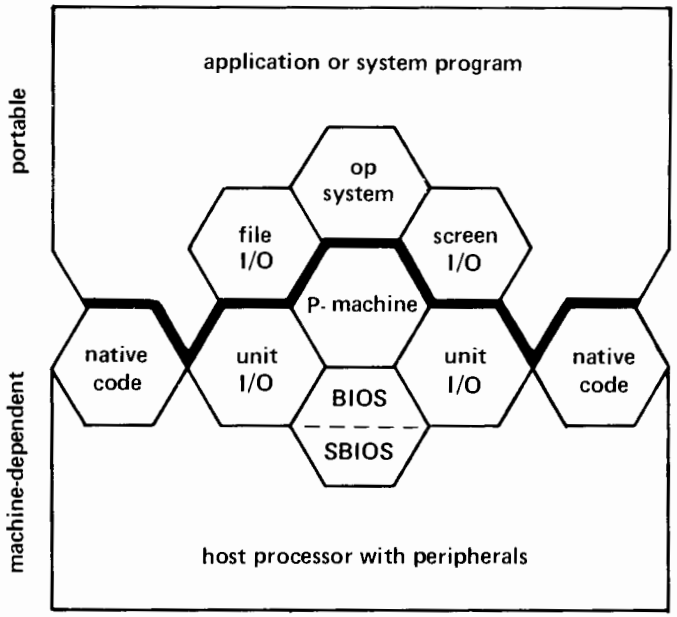
The System's standard input and output come from CONSOLE:. A user sits at the console, types commands and other input, and watches the console's screen for promptlines and other information from the System. The Filer can communicate with other devices, and so can a user's program (either using a language's standard I/O routines, or using special p-System intrinsics which can be much more efficient).

It is also possible to temporarily redirect the input or output of a program or the System itself: using either files other than the standard ones, or scratch buffers in main memory. This feature allows programs to be used as file "filters", and programs or the System itself to be driven by script files (a useful test tool). Refer to the e(Xecute and M(onitor commands in Chapter II.

To complete this section, Figure 3 shows a typical hardware configuration, with device names and a sketch of the Operating System's I/O interface.

Note: before the term UNIT was used in our System to denote a separately compiled portion of a program, devices/volumes were often called units as well. We have tried to rectify our terminology, but certain device-handling intrinsics are still named UNITREAD, UNITWRITE, and so forth. You should understand that these refer to device control and have nothing to do with program structure (discussed below). This confusion is unfortunate, but to change the names of the intrinsics would invalidate many programs now running in the field.

UCSD PASCAL I/O HIERARCHY



A SAMPLE SYSTEM

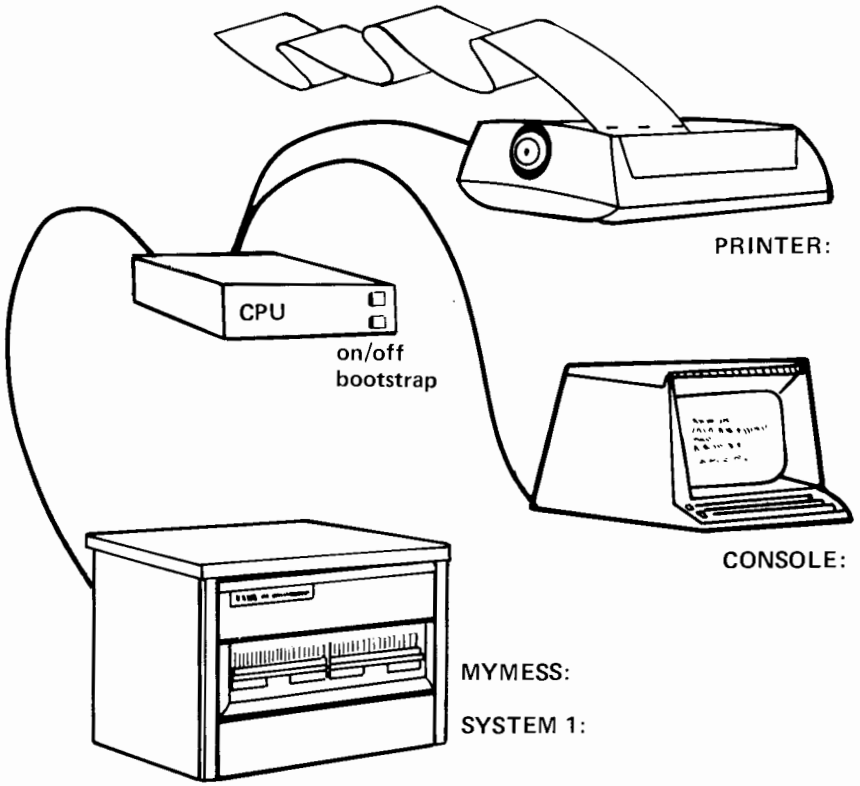


FIGURE 3

1.2.4 Program and Codefile Organization

A reasonably long program can fit into a single text file, be compiled in one piece, and executed as one block of code. But since many users require programs of substantial size, and since the UCSD p-System is a system for microprocessors, which have limited storage, it is frequently necessary to break a program up and compile it in two or more pieces.

There are other advantages to separate compilation. A single procedure may be used by several different programs, and so it might be most convenient to compile that procedure once and use it several times. The same might be true of a collected set of procedures, or some particular data structure. Judicious use of separate compilation can contribute to the organization of a large programming project.

The $Include option of the Compiler allows a programmer to store parts of program text in separate files. The Compiler reads them and compiles the entire program at one time. This is often a useful thing to do, especially if the included portions are not too long, and shared by more than one program. But using $Include does not address the problem of creating a program which is too large to compile in one piece.

Furthermore, it may be advantageous to embed procedures of a different language within a host program. This is the case when a program is not generally time-critical, but contains some time-critical sections -- the real-time sections may be isolated and written as assembly language routines.

This section and the rest of the Manual use the term "routine" to mean a procedure, function, or process, and the term "compilation unit" to refer to a program or UNIT. A compilation unit which uses separately compiled routines is called a "host compilation" or "client".

Note: Though this section uses Pascal for its program examples, the separate compilation and memory-management features available in Pascal have their analogs in the other high-level languages provided with the p-System. See documentation for the appropriate language.

A UNIT is a collection of routines and data structures. It may also contain initialization and termination code. Like a program, it may be compiled by itself, but unlike a program, it cannot be executed, except when invoked from a program. Programs and other UNITs may use UNITs that have already been compiled.

In the p-System, a codefile is organized into "segments". A compilation unit contains at least one segment -- the routines and data of the compilation unit itself. This segment is called the "principal segment". If the compilation unit contains SEGMENT routines (see below), each segment routine will be a "subsidiary

segment" that accompanies the principal segment. If the compilation unit references separately compiled UNITs, those are not considered subsidiary segments, but are named in a list of segment references that accompanies the principal segment. Segments are the basic unit of transfer when code is read from a disk or removed from memory.

The utility LIBRARY may be used to group compilation units together in a single codefile, and modify the organization of existing codefiles. Codefiles are often referred to as "libraries", especially when they don't contain a program.

When a host program that uses other units is executed, the System searches for the proper code, using the host segment's segment reference list.

The user may maintain one or more "library text files", which are files that contain a list of codefiles that a host compilation may need. When the System searches for a needed unit, it looks first (in order) at the codefiles named in the user's default library text file, and if that search fails, it looks in *SYSTEM.LIBRARY. The default name for the user's library text file is *USERLIB.TEXT; this can be changed by an execution option (see Chapter II). A compilation unit can also specify the library it needs by using the $U Compiler option. Libraries are discussed in detail in Chapter VII.

In the source code, a "client" compilation unit specifies that it needs a certain UNIT (or more UNITs) by a declaration immediately after the program (or UNIT) identification. For example:

```
PROGRAM W_CONTROL;
USES  SYNCHPROCS, TREES;
```

A UNIT itself may be outlined in the following way:

```
UNIT  I_AM_A_SAMPLE;

  INTERFACE
  ... {data declarations and procedure declarations}

  IMPLEMENTATION
  ... {data declarations and procedure code}

  begin  {initialization and termination block}
    ... {initialization code}
    ***;
    ... {termination code}
  end.
```

There are two main parts. The INTERFACE part contains declarations of procedures and data that may be used by the client. The IMPLEMENTATION part contains code for the procedures declared in the INTERFACE part, as well as data declarations and other procedures that are used by the procedures declared in the INTERFACE part, but which may not be used by the client. Finally, there is an optional section of Pascal code which contains two parts: an initialization part, which is code that is executed before any of the main body of the host program is executed, and a termination part, which is code executed after the host program's code has completed. These two parts are separated by '***;'.

When routines are assembled rather than compiled, they are declared EXTERNAL in the host program, e.g.:

```
PROCEDURE HANDSHAKE (VAR WHICH: STRING; SEM: INTEGER); EXTERNAL;
```

The assembled routines must carefully adhere to Pascal's calling and parameter-passing conventions, and respect System constraints on the use of machine resources such as registers. See Chapter VI.

External routines (assembled code) must be bound into a host by the Linker; once bound in, they remain part of the program. If the host program uses external routines contained in codefiles other than SYSTEM.LIBRARY, the Linker must be run explicitly (using the L(ink command).

To partition a program or UNIT into separate pieces that are independently loaded from disk as needed, the user may designate routines as SEGMENT routines, for example:

```
SEGMENT PROCEDURE FILL_CORE;
SEGMENT FUNCTION MUDDLE ( MEDDLE, MIDDLE: INTEGERS ): REAL;
SEGMENT PROCESS RUNAWAY (LOCK_IT: SEMAPHORE);
```

Each segment routine occupies one subsidiary segment in a codefile.

While a program is running, all code segments, both principal and subsidiary, compete for main memory on a dynamic basis. (The one exception to this is native code segments, which may have to be memory-resident. See Chapter VII.) Segments are loaded only when they need to be executed. When they are no longer needed, they remain in memory until the space they occupy is needed for some other use.

Using segment routines allows the System to better allocate memory, since only those segments that are being used need be in memory at any given time. The intrinsics MEMLOCK and MEMSWAP can be used to directly control the residence of a segment (see both the Pascal Reference Manual and the Internal Architecture Guide).

Such things as a program's routines for initialization and termination are prime candidates for declaring as SEGMENTs, since they are often bulky, and are called only once. There is no need for them to take up memory space after (or before) they have served their purpose.

Programs may be "chained", that is, a program may designate another program to be executed when the "chaining" program has finished executing. See the intrinsic CHAIN in the Pascal Reference Manual.

Using the p-System, standalone assembly language programs can be created, linked, loaded, and run. See Chapter VI on assemblers, and Section IX.1 on the COMPRESS utility.

A fuller discussion of the questions of separate compilation, linking, and memory management, is given in Chapter VII.

II. THE SYSTEM COMMANDS

This chapter includes a discussion of the commands at the System level, and a full description of each command. This is the outer level of System control, and these commands invoke basic System functions such as calling the Compiler, the Editor, the Filer, etc.

You may think of the System command level (the "outer" level) as the chief control for the entire System, which indeed it is -- you have already (in Figure 1) seen the System diagrammed as a tree of command levels, with the System commands as the outer level available from the root node.

It is also convenient, and in some ways more useful, to think of the System level as the communications interface between the sub-modules. Thus, the Filer initializes a workfile which the Editor uses to create a textfile which the Compiler uses to create a piece of a program which the Linker uses to create a runnable file which the eX(ecute command sets into operation. This sequence of events is controlled by the System commands. It is done "by hand", since the System was from the start conceived as an interactive environment. The point is that the System commands are what you must use to accomplish interaction between the various System components.

II.1 Promptlines

The promptline (sometimes called a menu) shows the command options at any given level of the System. Each command is invoked by a single letter -- 'E' for Edit, 'S' for Save, and so forth. Some things all promptlines have in common are:

...the name of the 'level' or System module at the beginning;

...a list of available commands, with the calling letter capitalized and separated from the rest of the word by '(';

...the version number of the program at the end of the line, in square brackets.

Here are a few representative promptlines:

Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem, D(ebug,? [IV.12 A]

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, ? [D.7]

>Edit: A(djust C(opy D(el F(ind I(nsert J(ump K(ol M(argin P(age ? [IV.1 F6c]

Anywhere in the p-System, a promptline will almost always be displayed at the top of the screen, and let you know what your options are. It is not always visible when you are using the Editor to insert text, and it is never visible while a user program is running. Typing unintelligible commands at any level may cause the promptline to go away; in this case, a space (' ') will cause the screen to be cleared and the correct promptline to be displayed.

Some promptlines include a '?'. There are often more commands than can fit onto one line, and typing '?' will display those commands. For example:

Command: H(alt, I(nitialize, U(ser restart, M(onitor [IV.12 A]

... is the remainder of the Command menu.

At the System command level, typing a command letter does one of four things: it calls a program such as the Filer, it does an operation on the workfile (such as C(ompile or R(un), it begins a file operation which will prompt you directly for one or more filenames (such as eX(ecute or L(ink), or it somehow alters the System state (such as H(alt). When you are prompted for filenames, you can usually omit the conventional suffix. For example, you wish to run GRISWICH.CODE. Type 'X' for eX(ecute. You will then be prompted:

Execute what file?

... and you type 'GRISWICH'<return>. The '.CODE' will be assumed.

The System will display as much of a promptline as will fit on a line, and typing '?' causes it to display the remaining commands.

II.2 Disk Swapping

Since the IV.1 Operating System does a good deal of swapping code segments into and out of main memory during the execution of a program, and since the user may change disks at various times (especially while running the System itself), the Operating System has various checks to aid disk handling, and reduce the possibility of error.

When a program requires a code segment that is on disk, and it is no longer on the disk in the drive from which it was originally read, the Operating System will display a prompt that looks something like this:

Need segment SEGSCCHE: Put volume USER1 in unit 4 then type <space>

... in this example, the System requests the disk USER1:, and will wait until the user types <space>. (If the user types <space> but has not replaced USER1:, the System will redisplay the prompt.)

If at any time during the execution of a program, a device is found to contain a volume that the System did not expect, the System considers that device "questionable" for the remainder of that program's execution. All subsequent reads and writes that the Operating System does to that device will check to see that the volume name is correct (provided the correct volume name is known). If the volume name is deemed incorrect, the System displays a prompt of the following sort:

```
Put volume USER2 in device # 5
type <space> to continue
```

... in this case, the System expected the disk USER2:, did not find it, and therefore requested it.

These situations should not often arise, but will occur when a program requires more disk storage space than is available from on-line disk drives.

This sort of checking is not done for explicit UNITREADs and UNITWRITEs that may appear in a user program.

II.3 Execution Option Strings

The eX(ecute command allows the user to specify some options that modify the System's environment. These include redirecting standard program I/O or standard System I/O, changing the default prefix (i.e., the volume name part of a filename; see Chapter III or Section I.2.3), and changing the default library text file (see Section I.2.4 or Chapter VII. These options are also available from within a user program.

All of these options are specified by means of "execution option strings". An execution option string is a string that contains (optionally) one filename, and zero or more option specifications. An option specification consists of one or two letters followed by an equals sign ('='), possibly followed by a filename or literal string.

These are the possible execution options, with a summary of their uses:

L=	change the default library text file
P=	change the default prefix
PI=	redirect program input
PO=	redirect program output
I=	redirect System input
O=	redirect System output

... either capital or lower-case letters may be used.

Several different execution options may be entered at a single time. If this is the case, they must be separated by one or more spaces. There may optionally be a single space between the '=' and the following filename or string.

These options are described in full detail below. They may be invoked by using the eX(ecute command.

Redirecting System input to come from a file or main memory amounts to driving the System from a script of commands. This is a useful tool, especially in testing or turnkey applications. One way to create a script for the System is to use the M(onitor command, which records keystrokes by writing them to a file while they are performed. M(onitor is described in Section II.4.9.

Note: Redirection applies only to the standard files 'input' and 'output', and therefore has no effect on low-level device I/O intrinsics such as UNITWRITE, BLOCKREAD, etc.

11.3.1 Alternate Prefixes and Libraries

The user can change the default prefix with the P= execution option string. After this is done, all filenames that do not explicitly name a volume will be prefixed by the default prefix. This is equivalent to using the P(refix command in the Filer (see Chapter 111).

Example (user inputs are underlined):

Type 'X':

Execute what file? p=zoom

... the default prefix is now ZOOM:.

In a similar fashion, the default "library text file" can be changed. The library text file is a file that contains the names of a number of user libraries. When a program with separately compiled units is run, the System searches for them first in the files named in the library text file, and then in *SYSTEM.LIBRARY. When the System is booted, the default library text file is *USERLIB.TEXT. More information about libraries may be found in Chapter VII .

To change the default library text file, use the execution option string L=.

Examples:

Execute what file? L=mylib

... makes the file MYLIB.TEXT the new default library text file.

Execute what file? advent l=mylib

... makes the file MYLIB.TEXT the new library text file, and executes the file ADVENT.CODE.

Important note: The order in which execution options are performed is:

1) Change prefix (if the P= option is present);
2) Change library text file (if the L= option is present);
3) Do the I/O redirections (if any present)(the order of redirection options is irrelevant).

11.3.2 Redirection

The following execution option strings control redirection:

```
PI=<filename>
PI=<string>
PO=<filename>
I=<filename>
I=<string>
O=<filename>
```

PI= redirects program input. PI=<filename> causes the input to a program to come from the file named. PI=<string> causes the input to a program to come from the program's scratch input buffer, and appends the string given to the scratch input buffer (scratch input buffers are discussed below).

PO= redirects program output. PO=<filename> causes program output to be sent to the file named.

PI= overrides any previous input redirection. Likewise, PO= overrides any previous output redirection. Using PI= (PO=) without a filename makes program input (output) the same as System input (output).

I= redirects System input. I=<filename> causes System input to come from the file named. I=<string> causes System input to come from the System's scratch input buffer, and appends the string to the scratch input buffer.

O= redirects System output. O=<filename> causes System output to be sent to the file named.

Like PI=, I= overrides any previous I=, and like PO=, O= overrides any previous O=. Using I= without a filename resets System input to CONSOLE:. Using O= without a filename resets System output to CONSOLE:.

For PI=<filename> and I=<filename>, the <filename> may specify either a disk file or an input device that sends characters. If the file is a disk file, redirection ends at EOF; the System performs the equivalent of an input redirection with no filename, thus resetting input. If the file is a device, redirection continues until explicitly changed by the user. This allows a user to control the System from a remote port (such as REMIN:).

For PO=<filename> and O=<filename>, the <filename> may specify either a disk file or an output device that receives characters. If the file is a disk file, it is named literally as shown (i.e., to make it a textfile, the user must explicitly type .TEXT). Whenever output redirection is changed, the file is closed and locked.

For P1=<string> and 1=<string>, the <string> may be any sequence of characters enclosed in double quotes ('"'). Any double quote embedded in the string must be typed twice. Scratch input buffers are located in main memory. Program or System input may be redirected to come from both a file and the appropriate scratch input buffer, but if this is the case, the scratch buffer will be used first (until it is empty). Strings are always appended to scratch input buffers, so that they are read in order (i.e., first in, first out). Commas in scratch input buffers are treated as carriage returns (<return>).

Program redirection ends when the program terminates. If there are still characters in the program's scratch input buffer, they are lost.

System redirection ends when the System terminates with a H(alt command, an I(nitialize command, or a runtime error.

Note that redirection applies only to the standard files called 'input' and 'output' in Pascal (which have their analogs in the p-System's other high-level languages).

Examples:

Execute what file? YEEN PI=IN PO=OUT

Execute what file? PO= OUT PI= IN yeen

... both redirect program input to the file IN, and program output to the file OUT. The program is YEEN.CODE.

Execute what file? I=

... stops System input redirection.

Execute what file? PO= storeme.text PI= I="fgRUNME,qr" P=WORK2

... this would:

make the default prefix WORK2:;
redirect program output to the file WORK2:STOREME.TEXT;
turn off program input redirection;
cause the System to follow the script "fgRUNME,qr", which would:
f: enter the Filer;
gRUNME,: G(et the workfile WORK2:RUNME.TEXT
and WORK2:RUNME.CODE;
(note that the comma acts as a carriage return)
q: Q(uit the Filer, and ...
r: R(un the program WORK2:RUNME.CODE
(note that its output has been redirected).

... this was admittedly an elaborate, though not inconcievable, example. The following is a slightly different example that would do the same thing:

Execute what file? PO= storeme.text PI= I="fpWORK2:,gRUNME,qr"

... although using the Filer to change the default prefix is probably a waste of time and space.

11.4 Individual Commands Alphabetically

11.4.0 Prompts for Filenames

Several of the System commands prompt for filenames. The conventions are the same for all responses to filename prompts throughout the System. A filename is typed in as letters, and followed by a <return>. Before <return> is typed, the name may be corrected by using <backspace> or <delete line> and re-typing. Prompts often expect .TEXT or .CODE files, and these standard suffixes may be omitted from the filename -- the System programs will append them automatically. To prevent this automatic appending, follow the filename with a '.'.

When a program (such as a compiler) requires both a source and a code file name, the codefile name may be given as '$', which is the same name as the source file with .CODE appended, or as '$.', which is the source file name only.

Example (underlined portions are user input):

```
Assemble what text? GRISWICH
To what code file? $
```

... causes the file GRISWICH.TEXT to be assembled, and the resulting code placed in GRISWICH.CODE.

Device names may also be used.

Example:

```
Output file for assembled listing: (<cr> for none)
```

Responding to a filename prompt with just <return> causes some default filename to be used (e.g., *SYSTEM.WRK.CODE). If there is no default value, the program will go on to the next action (or abort, because there is nothing left for it to do).

11.4.1 ASSEMBLE

On the promptline: A(ssem.

Causes SYSTEM.ASSMBLER (note no 'E') to be executed. If a workfile is present, then either *SYSTEM.WRK.TEXT or the designated .TEXT file is assembled to a file of HP-86/87 native code. If there is no workfile, the user is prompted for a source file. The user is also prompted for a codefile and a listing file; the defaults for these are *SYSTEM.WRK.CODE and no listing file.

If the Assembler encounters a syntax error, it displays the error number, the source line in question, and (if the file *HP86/87.ERRORS is present) an error message; finally, it displays the promptline:

Line ##, error ###: <sp>(continue), <esc>(terminate), E(dit

The user has the choice of continuing assembly (<space>), aborting assembly (<escape>), or returning directly to the Editor to correct the source file ('E').

Chapter VI describes the Assembler in detail.

11.4.2 COMPILE

On the promptline: C(omp.

Causes SYSTEM.COMPILER to be executed. If a workfile is present, then either *SYSTEM.WRK.TEXT or the designated .TEXT file is compiled to P-code. If there is no workfile, the user is prompted for a source file. The user is also prompted for a codefile name; the default for this is *SYSTEM.WRK.CODE.

If the Compiler encounters a syntax error, it displays the error number, the source line in question, and the promptline:

Line ##, error ###: <sp>(continue), <esc>(terminate), E(dit

The user has the choice of continuing compilation, aborting compilation, or returning directly to the Editor to correct the source file. In the latter case, the cursor will be positioned at the point of error detection, and an error number or message will be displayed.

FORTRAN and Pascal are described in separate manuals.

II.4.3 DEBUG

On the promptline: D(ebug

Causes the Debugger in SYSTEM.PASCAL to be executed. The Debugger is used to "freeze" the execution of a user program so that the p-Code can be disassembled, the contents of memory addresses can be examined and modified, and the execution can be single-stepped.

The Debugger is described in detail in Chapter IX.

II.4.4 EDIT

On the promptline: E(dit.

Causes SYSTEM.EDITOR to be executed. If a .TEXT workfile is present, this is displayed and available for editing. If no workfile is present, the user is prompted for a filename, with the additional options of either <esc>aping the Editor, or entering the Editor with no file at all (with the intent of creating a new one).

The Editor is used for creating program or document textfiles, or altering and adding to existing ones. It is described in detail in Chapter IV. Some users use YALOE as SYSTEM.EDITOR; YALOE is described in Chapter V.

II.4.5 FILE

On the promptline: F(ile.

Causes SYSTEM.FILER to be executed. The Filer provides commands for maintaining the workfile, moving files, and maintaining disk directories. It is described in detail in Chapter III.

II.4.6 HALT

On the promptline: H(alt.

Causes the System to stop p-System execution and return control to the HP-86/87 operating system.

II.4.7 INITIALIZE

On the promptline: I(nit.

Causes the file *SYSTEM.STARTUP, if present, to be executed. SYSTEM.STARTUP must be a codefile; it is executed automatically after a bootstrap or an 'I' command.

You may create your own SYSTEM.STARTUP. Some applications of this might be displaying reminders for the next session with the System, or creating a program to run in a turnkey mode. To create a SYSTEM.STARTUP, you must create a .CODE file, and then change its name to SYSTEM.STARTUP.

All runtime errors that are not "fatal" (see Appendix A) cause the System to do an initialize. At initialize time, much of the System's internal data is rebuilt, and SYSTEM.MISCINFO is reread.

II.4.8 LINK

On the promptline: L(ink.

Causes the file SYSTEM.LINKER to be executed. The Linker allows you to link native code (assembled) routines into host compilation units (compiled from a high-level language). It also allows you to link native code routines together. It is described in detail in Chapter VII, particularly Section VII.4.

II.4.9 MONITOR

On the promptline: M(on.

Redirecting the System's input (see Section II.3) amounts to driving the System with a script; one convenient way to create such a script is to use M(onitor. While in M(onitor mode, the user may use the System in a normal manner, but all user input is saved in a file. Thus, to automate a sequence of System commands, the user B(egins a monitor, and goes through all the commands that are to be remembered. Then the user E(nds the monitor, and all user input is saved as a file. This file can be used by redirecting System input to the monitor file with the I= execution option string.

When 'M' is typed to enter M(onitor, the following prompt is displayed:

Monitor: B(egin, E(nd, A(bort, S(uspend, R(esume

B(egin starts a monitor. The user is prompted for a filename, and then returned to the System promptline. If a monitor file has already been opened, an error message is displayed.

E(nd ends a monitor and saves the monitor file.

A(bort ends a monitor but does not save the monitor file.

S(uspend turns off monitoring but does not close the monitor file. In other words, the user is returned to the System promptline and can now type commands without recording them, but the monitor file remains open, and more can be added to it by using R(esume.

R(esume starts monitoring again, and returns the user to the System promptline. After a monitor has been ended, R(esume displays "no monitor open," returns the user to the System promptline, and enables the user to execute the monitor file.

The monitor file can be either a .TEXT file or a datafile. If it is .TEXT, the user can use the Editor to alter it -- but not if the monitoring has recorded special characters which the Editor does not allow a user to type.

The M(onitor command itself can never be recorded in a monitor file.

II.4.10 RUN

On the promptline: R(un.

Causes the current workfile to be executed. If there is no current codefile in the workfile, R(un calls the Compiler, and if the compilation is successful, runs the resulting code. If there is no workfile at all, R(un calls the Compiler, which then prompts for the name of a textfile to compile.

If the codefile requires linking to one or more external codefiles, then the Linker is automatically called, and searches *SYSTEM.LIBRARY. If the external files cannot be found there, an error results.

II.4.11 USER RESTART

On the promptline: U(ser restart.

Causes the last program executed to be executed over again, with all file parameters equal to what they were before. U(ser restart will not restart the Compiler or Assembler. Other than that, it is useful for multiple runs of a user program, returning to the Editor after a workfile U(pdate, and so forth.

II.4.12 EXECUTE

On the promptline: X(ecute.

eX(ecute displays the following prompt:

Execute what file?

... and the user should respond with an execution option string (see Section II.3, above). In the simplest case, this string contains nothing but the name of a codefile to be executed (as described in Section II.4.0).

If the codefile cannot be found, the message 'Can't find file' is displayed. If all the code necessary to execute the codefile has not been linked in, the message 'Must L(ink first' is displayed. If the codefile contains no program (i.e., all its segments are units or segment routines), the message 'No program in '<filename> is displayed.

If the execution option string contains only option specifications, they are treated as described in Section II.3, above. If it contains both option specifications and a codefile name, the options are handled first, and then the codefile is executed (unless one of the errors named in the preceding paragraph occurs).

eX(ecute is commonly used to call programs that have already been compiled. It may also be used simply to take advantage of the execution options.

The codefile must have been created with a .CODE suffix, even if its name has subsequently been changed.

III. FILES AND FILE HANDLING

III.1 Types of Files

A file is a collection of information which is stored on a disk and referenced by a filename. Each disk has a directory which contains the filename and location of each file on the disk. The Filehandler, or Filer, uses the information contained in the disk directory to manipulate files.

One of the attributes of a file is its type. The type of the file determines the way in which it can be used. Filetypes are indicated by the suffix to the filename (if one is present; the directory maintains a filetype field for each file). Reserved type suffixes for filenames are:

.TEXT .BACK	Human readable text, formatted for the editors.
.CODE	Executable code, either P-code or machine code.
.DATA	Data in a user-specified format.
.FOTO	A file containing one graphic screen image.
.BAD	An unmovable file covering a physically damaged area of a disk.

III.2 File Formats

.TEXT and .BACK files contain a header page followed by the user-written text, interspersed with blank-compression codes. The header page contains internal information for the editors. The Filer will transfer the header page from disk to disk, but never from disk to an output device (e.g., PRINTER: or CONSOLE:).

Note that all files created with a suffix of .TEXT will have the header attached to the front, and so they will be treated as textfiles throughout their life.

The header page is two blocks long (1024 bytes), and the remainder of the file is also organized into two-block pages. A page contains a series of complete text lines, and is padded with NULs. A complete text line is 0..1024 characters -- the last of those characters must be a <return> (ASCII CR), and the first two may be a blank-compression pair. The optional blank-compression pair consists of an ASCII DLE followed by a byte whose value is 32+n, where n is the number of characters to indent. Text lines are typically 0..80 characters in length, so as to fit on standard terminals.

Textfiles are considered to be unstructured files, and so the intrinsic SEEK will not work with them. (SEEK is described in the Pascal manual.)

.CODE files contain either compiled or assembled code. They begin with a single block called the segment dictionary, which contains internal information for the Operating System and Linker. Codefiles may also contain embedded information. They are described in detail in the Internal Architecture Guide.

.DATA files have any format that their creator chooses. The System knows nothing about the internals of a datafile.

.FOTO files are declared as follows:

```
type screen = packed array [0..239, 0..319] of Boolean;
                (* These dimensions are for Graph_Mode, the default. *)

                                --or--

              packed array [0..239, 0..543] of Boolean; (* Graph_All_Mode *)

var fotofile: packed file of screen;
```

.FOTO files are created with the Turtlegraphics routines discussed in Chapter IX.

III.3 Volumes

A volume is any I/O device, such as the printer, the keyboard, or a disk. A block-structured device is one that can have a directory and files, usually a disk of some sort. A non-block-structured device does not have internal structure; it simply produces or consumes a stream of data. The printer and the keyboard, for example, are non-block-structured. The table below illustrates the reserved volume names used to refer to non-block-structured devices, the device number associated with each device, and the device names associated with the System disk and other peripherals.

Device Number	Volume ID	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTERM:	screen and keyboard without echo
4	SYSTEM:	the System disk
5	<volume name>:	the alternate disk
6	PRINTER:	the line printer
7	REMIN:	serial line input
8	REMOUT:	serial line output
9-22	<volume name>:	additional disk drives
23	RAMDISC:	electronic disk (RAM > 160K bytes)

TABLE 111.1

111.4 The Workfile

The workfile is described in Section 1.2.2.3. It is a 'scratchpad' for creating files, and testing those files if they contain program text. The workfile is often stored temporarily in the files SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE. These may be either newly-created files, or copies of existing disk files which have been designated as the new workfile.

The Filer is the means of saving a workfile under permanent filenames (the S(ave command), designating existing files as the current workfile (the G(et command), or clearing a workfile for new work (the N(ew command). More detail on these functions is provided in the description of each of these commands, and you should refer to those discussions below.

III.5 Filenames

Many Filer commands, System prompts, and System Intrinsics require the user to respond with at least one file specification. The diagram below illustrates the syntax of file specification.

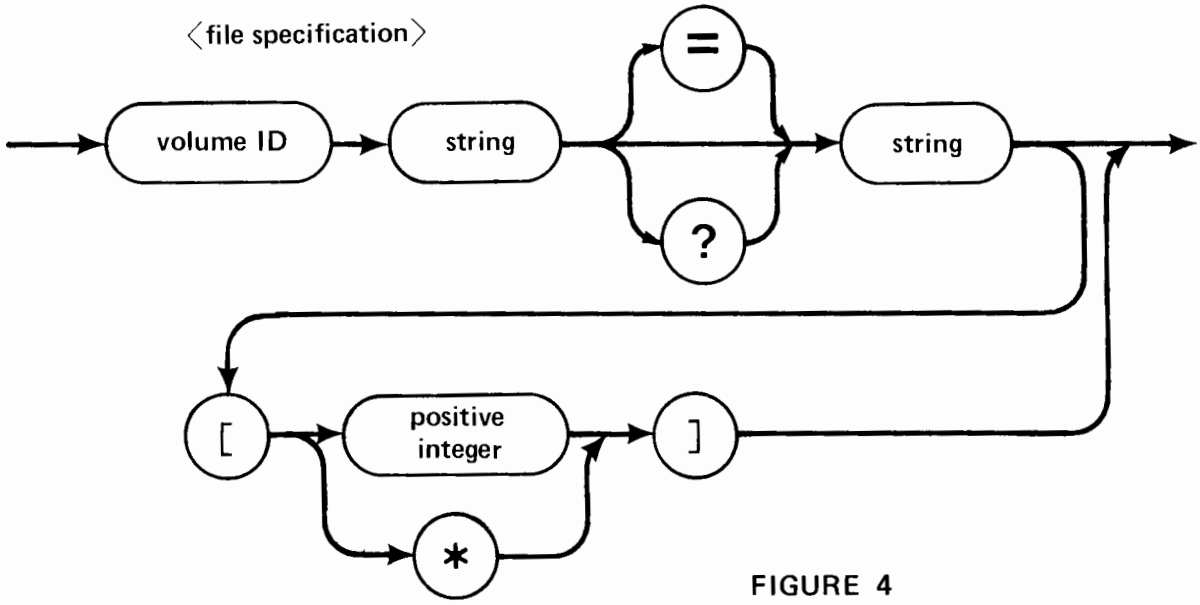


FIGURE 4

Volume ID syntax can be expanded thus:

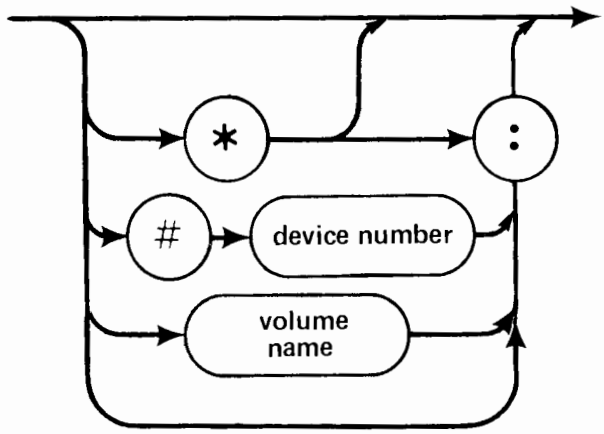


FIGURE 5

As shown in the table above, the volume name (e.g., 'CONSOLE:') and the physical device number (e.g., '#1:') may both be used, and are in fact interchangeable.

Volume names for block-structured volumes can be assigned by the user. A volume name must be 7 characters long or less and may not contain '=', '$', '?' or ','. Reserved volume names for non-block-structured devices are given in Table III.1. The character '*' is shorthand for the volume ID of the System disk. The character ':' is shorthand for the volume ID of the default disk. The System disk and default disk are equivalent unless the default prefix has been changed. This can be done with the P(refix command (see below). The System disk is also called the 'root disk' here and there. '#<device number>' is equivalent to the name of the volume in the drive at that time.

A legal filename can consist of up to 15 characters, including the .TEXT and .CODE suffixes, which are appended to a filename when the file is created, and reflect the internal organization of the file. Lower-case letters are translated to upper-case, and blanks and non-printing characters are removed from the filename. Legal characters for filenames are the alphanumerics and the special characters '-', '/', ' ', '_', and '.'. These special characters may be used as mnemonics to indicate relationships among files and/or to distinguish several related files of different types.

Filenames must not contain the following special characters: '$', ':', '=', '?', and ','. The reason will become apparent in the next section.

III.6 Using the Filer

Filer commands are described in detail below, in section III.6.3. They are listed one to a page, in alphabetical order. It is recommended that you read the following two sections as background for using the Filer commands; this entire chapter is meant to serve both as instruction and as a reference.

III.6.1 Prompts in the Filer

Type "F" at the Command level to enter the Filer. The following prompt is displayed:

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, ? [D.7]

Typing '?' once and again in response to this prompt displays the remaining two lines of Filer commands.

The individual Filer commands are invoked by typing the letter found to the left of the parenthesis. For example, 'S' would invoke the Save command.

In the Filer, answering a Yes/No question with any character other than 'Y' constitutes a 'No' answer. Typing an <esc> will return the user to the outer level of the Filer.

Many commands will prompt you for a filename. The full syntax for a file specification (which is either a single filename or an expression using wildcards) is given in Figures 4 and 5 above. Always follow file specifications with a <return>. For a description of wildcards, see below.

Should you specify a file on a volume (or just a volume) that the Filer cannot find, it will respond with:

No such vol on-line

If more than one volume on line has the same name, the Filer will continually display a warning to that effect. The user must be careful to specify which volume a file is on (usually using device numbers, e.g., #4, #5) in order to avoid confusion. The situation is especially confusing when both disks are System disks. In general, although it may sometimes be necessary to have two volumes with the same name on line together, the user should try to avoid this situation.

Whenever a Filer command requests a file specification, the user may specify as many files as desired, by separating the file specifications with commas, and terminating this 'file list' with a <return>. Commands operating on single

filenames will keep reading filenames from the file list and operating on them until there are none left. Commands operating on two filenames (such as C(hange and T(rans) will take file specifications in pairs and operate on each pair until only one or none remains. If one filename remains, the Filer will prompt for the second member of the pair. If an error is detected in the list, the remainder of the list will be flushed.

III.6.2 Names of Files

III.6.2.1 General Filename Syntax

For the Filer, filename syntax is the same as for the System in general, as described above in Section III.5. In addition, a filename may be followed by a size specification of the form '[n]' where n is an integer specifying the number of blocks that the file must occupy. Size specifications are dealt with below, in the description of those commands that are affected by them.

All of the Filer commands except G(et and S(ave require full filenames, including suffixes such as .TEXT and .CODE. G(et and S(ave supply these suffixes automatically, so that using the workfile will be convenient.

III6.2.2 Wildcards

The wildcard characters, '=' and '?', are used to specify subsets of the directory. The Filer performs the requested action on all files meeting the specification. A file specification containing the subset-specifying string 'DOC=TEXT' notifies the Filer to perform the requested action on all files whose names begin with the string 'DOC' and end with the string 'TEXT'. If a '?' is used in place of an '=', the Filer requests verification before performing the command on each file meeting the specified criteria. A subset specification of the form '=<string>' or '<string>=' or even '=' is valid. This last case, where both subset-specifying strings are empty, is understood to specify every file on the volume, so typing '=' or '?' alone causes the Filer to perform the appropriate action on every file in the directory.

EXAMPLE:

Given this directory for the volume MYDISK:

NAUGHTYBITS	6	23-Jun-54
MOLD.TEXT	4	29-Jun-54
USELESS.CODE	10	19-May-54
MOLD.CODE	4	29-Jun-54
NEVERMORE.TEXT	12	5-Apr-54
GOONS	5	10-Sep-52

Prompt: Remove what file?

Response: Typing 'N=' generates the message:

```
MYDISK:NAUGHTYBITS          removed
MYDISK:NEVERMORE.TEXT       removed
Update directory?
```

At this point the user can type 'Y' to remove or type 'N', in which case the files will not be removed. (The Filer always requests verification on removes.)

Typing 'N?' generates the message:

Remove NAUGHTYBITS: ?

After the user types a response, the Filer asks:

Remove NEVERMORE.TEXT: ?

EXAMPLE:

Prompt: Dir listing of what vol ?

Response: Typing '=TEXT' causes the Filer to list

```
MOLD.TEXT                    4  29-Jun-54
NEVERMORE.TEXT              12   5-Apr-54
```

The subset-specifying strings may not 'overlap'. For example, GOON=NS would not specify the file GOONS, whereas GOON=S would be a valid (although pointless) specification.

In any filename pair, the character '$' may be used to signify the same filename as the first name, perhaps with a different volume id or size specification.

EXAMPLE:

Prompt: Transfer what file?

Response: '#5:RE.USE.TEXT,*$'

... transfers the file RE.USE.TEXT on device #5 (a disk drive) to the System disk ('*', which is also device #4). The name is not changed. The Filer would reply with:

```
WORKSET:RE.USE.TEXT                    ->   SYSTEM:RE.USE.TEXT
```

III.6.3 Filer Commands

This section contains complete descriptions of all Filer commands, together with examples of their use. Commands are listed in alphabetical order, each new command beginning on a new page. The text is meant to be used both as instruction and as a reference.

B(ad blocks

Reference Manual
File Handling

111.6.3.1 B(ad blocks

Scans the disk and detects blocks that are unusable for some physical reason (fingerprints, warping, dirt, etc.).

This command requires the user to type a volume ID. The specified volume must be on-line.

Prompt: Bad block scan of what vol?

Response: <volume ID>

Prompt: Scan for 520 blocks ? <y/n>

Response may be "Y" for yes if you want to scan for the entire length of the disk. If you only wish to check a smaller portion of the disk, type "N" and you will then be prompted for the number of blocks you want the Filer to scan for. The purpose of this part of the command is for disks where the Filer has no idea of how 'long' the device is.

Checks each block on the indicated volume for errors and lists the number of each bad block. Bad blocks can often be fixed or marked (see eX(amine).

III.6.3.2 C(hange

Changes file or volume name.

This command requires two file specifications. The first of these specifies the file or volume name to be changed, the second, the new name. The first specification is separated from the second specification by either a <return> or a comma (','). Any volume ID information in the second file specification is ignored, since obviously the 'old file' and the 'new file' are on the same volume! Size specification information is ignored.

Actual movement of files from volume to volume is done with the T(ransfer command.

Given the example file F5.TEXT, residing on the volume occupying device 5:

Prompt : Change what file?

User Response: #5:F5.TEXT,HOOHAH

... changes the name in the directory from 'F5.TEXT' to 'HOOHAH'. Filetypes are originally determined by the filename; the C(hange command does not affect the filetype. In the above case, HOOHAH would still be a textfile. However, since the G(et command searches for the suffix '.TEXT' in order to load a textfile into the workfile, HOOHAH would need to be renamed HOOHAH.TEXT in order to be loaded into the workfile.

The user response '#5:F5=,HOOHAH=', on the other hand would preserve the .TEXT suffix.

Wildcard specifications are legal in the C(hange command. If a wildcard character is used in the first file specification, then a wildcard must be used in the second file specification. The subset-specifying strings in the first file specification are replaced by the analogous strings (henceforth called replacement strings) given in the second file specification. The Filer will not change the filename if the change would have the effect of making the filename too long (>15 characters).

EXAMPLE:

Given a directory of example disk NOTSANE: containing the files:

```
POEMS.TEXT
MAUNDER.TEXT
MALPRACTICE
MAKELISTS.TEXT
```

Prompt : Change what file?

User response: NOTSANE:MA=TEXT,XX=GAACK
causes the Filer to report

```
NOTSANE:MAUNDER.TEXT          -> XXUNDER.GAACK
NOTSANE:MAKELISTS.TEXT        -> XXKELISTS.GAACK
```

The subset-specifying strings may be empty, as may the replacement strings. The Filer considers the file specification '=' (where both subset-specifying strings are empty) to specify every file on the disk. Responding to the C(hange prompt with '=,Z=Z' would cause every filename on the disk to have a 'Z' added at front and back. Responding to the prompt with 'Z=Z,=' would replace each terminal and initial 'Z' with nothing.

EXAMPLE:

Given the filenames:

```
THIS.TEXT
THAT.TEXT
```

Prompt : Change what file?

User Response: T=T,=

The result would be to change 'THIS.TEXT' to 'HIS.TEX', and 'THAT.TEXT' to 'HAT.TEX'.

The volume name may also be changed by specifying a volume ID to be changed, and a volume ID to change to.

EXAMPLE:

Prompt : Change what file?

User Response: NOTSANE:,WRKDISK:

NOTSANE: -> WRKDISK:

III.6.3.3 D(ate

Lists current system date, and enables the user to change the date.

```
Prompt:  Date Set: <1..31>-<JAN..DEC>-<00..99>
         Today is 19-Aug-78
         New date?
```

The user may enter the correct date in the format given. After typing <return>, the new date will be displayed. Typing only a return does not affect the current date. The hyphens are delimiters for the day, month and year fields, and it is possible to affect only one or two of these fields. For example, the year could be changed by typing '--79', the month by typing '-Sep', etc. The entire month-name can be entered, but will be truncated by the Filer. Slash ('/') is also acceptable as a delimiter. The most common input is a single number, which is interpreted as a new day. For example, if the date shown is the 19th of August, and today is the 20th, the user would type '20'<return>; this would have the desired effect of changing the date to the 20th of August. The day-month-year order is required.

This date will be associated with any files saved or created during the current session and will be the date displayed for those files when the directory is listed.

The date is saved in the directory of any disk that has been placed in the booted device. It remains the same until it is changed by using the D(ate command again.

III.6.3.4 E(xtended list

Lists the directory in more detail than the L(dir command.

All files and unused areas are listed along with (in this order) their block length, last modification date, the starting block address, the number of bytes in the last block of the file, and the filetype. All wildcard options and prompts are as in the L(dir command.

Since this command shows the complete layout of files and unused space on the disk, it is useful in conjunction with the M(ake command. Refer to Section III.6.3.9 and section III.6.4 on recovering lost files.

An E(xtended list is often longer than will fit on one screen. In this case, the Filer displays one full screen and then prompts:

Type <space> to continue

... at this point, a <space> causes the rest of the directory to be listed, and an <esc> aborts the listing.

EXAMPLE:

```
MYDISK:
FILERDOC2.TEXT     28    1-Sep-78      6        512      Textfile
ABSURD.CODE        18    1-Sep-78     34        512      Codefile
<UNUSED>           10                 52
ABSURD              4    1-Sep-78     62        512      Datafile
HYTYPER.CODE       12    1-Sep-78     66        512      Codefile
STASIS.TEXT         8    1-Sep-78     78        512      Textfile
LETTER1.TEXT       18    1-Sep-78     86        512      Textfile
ASSEMDOC.TEXT      20    1-Sep-78    104        512      Textfile
FILERDOC1.TEXT     24    1-Sep-78    124        512      Textfile
<UNUSED>          200                148
STASIS.CODE         6    1-Sep-78    348        512      Codefile
<UNUSED>          154                354
10/10 files<listed/in-dir>, 138 blocks used, 382 unused, 200 in largest
```

III.6.3.5 F(lip-swap/lock

This command can facilitate use of the filer on systems that have enough memory.

The Pascal code that makes up the filer is divided into several segments. Not all of the segments are needed at the same time. By removing unneccessary segments from memory, more memory space is available for the filer to perform its tasks. For example, a T(ransfer will work much more efficiently if there is a large buffer area available in memory. Furthermore, on some machines, there is simply not enough memory space to contain the entire filer.

However, allowing the filer to have nonresident segments requires that the disk containing SYSTEM.FILER be accessed whenever a non-resident segment is needed. This can be inconvenient on most two-drive systems. It would be more convenient to enter the filer, remove the system disk if desired, and perform any combination of L(isting, disk to disk T(ransferring, K(runching, etc., without having to replace the system disk at frequent intervals.

In the first mode, the filer segments are memswapped, and in the second mode, they are memlocked. The F(lip swap/lock command allows the user to make the choice of which mode the filer will use. The initial state upon entering the filer is always memswapped. Pressing "F" acts as a toggle between the memswapped and memlocked states. For example, if you entered the filer and pressed "F" twice, you would receive two prompts similar to the following:

```
Filer segments memlocked [9845 words]
Filer segments swappable [13918 words]
```

The number of available 16-bit words is given so that you will have an idea of how much space is left for the filer to perform its functions. There is usually less space available in the memlocked mode. If the machine does not have enough space to memlock the filer segments, the user will receive a message indicating that lack of space. (If there is not at least 1500 extra words available, the filer will not allow the memlock option.)

III.6.3.6 G(et

Loads the designated file into the workfile.

The entire file specification is not necessary. If the volume ID is not given, the default disk is assumed. Wildcards are not allowed, and the size specification option is ignored.

EXAMPLE:

Given the directory:

FILERDOC2.TEXT
ABSURD.CODE
HYTYPER.CODE
STASIS.TEXT
LETTER1.TEXT
FILER.DOC.TEXT
STASIS.CODE

Prompt: Get what file?

Response: STASIS

The Filer responds with the message

'Text & Code file loaded'

... since both text and code file exist. Had the user typed 'STASIS.TEXT' or 'STASIS.CODE', the result would have been the same -- both text and code versions would have been loaded. In the event that only one of the versions exists, as in the case of ABSURD, then that version would be loaded, regardless of whether text or code was requested. Typing 'ABSURD.TEXT' in response to the prompt would generate the message: 'Code file loaded'. Working with the file may cause the files SYSTEM.WRK.xxxx to be created, as part of the workfile. These files will go away when the S(ave command is used. If the System is rebooted before the S(ave command is used, the name of the workfile will be forgotten.

III.6.3.7 K(runch

Moves the files on the specified (disk) volume so that they are adjacent, and unused blocks are combined into one large area.

K(runch first prompts for the name of a volume. It then asks if it should crunch from the end of the disk. This leaves all files at the front of the disk, and one large unused area at the end. If the user answers no to this prompt, K(runch asks which block the crunch should start from. Doing a K(runch from a block in the middle of the disk leaves the large unused area in the middle of the disk, with files clustered toward either end (as space permits).

As each file is moved, its name is displayed on the console.

If the disk contains a bad block that has not been marked (see B(ad and eX(amine), K(runch may write a file on top of it -- that file is then irrecoverable. It is generally a good idea to scan for bad blocks with B(ad before doing a K(runch, unless all files are also backed up on a different disk.

If K(runch must move SYSTEM.PASCAL, it will then display a prompt which asks you to reboot the System. Do not do anything else until you have done so. If you do, the information on your disk may be irretrievably garbled.

EXAMPLE:

Prompt: Crunch what vol?

Response: MYDISK:

... if MYDISK: is on-line, K(runch then prompts:

Prompt: From end of disk, block 520 ? (Y/N)

Response: A 'Y' starts the K(runch, an 'N' causes the prompt:

Prompt: Starting at block # ?

Response: The block number at which you wish the K(runch to start.

III.6.3.8 L(dir

Lists a disk directory, or some subset thereof, to the volume and file specified (default is CONSOLE:).

Each filename is followed by the file's length in blocks, and the date of its last modification. (A block is 512 bytes).

The user may list any subset of the directory, using the wildcard option, and may also write the directory, or any subset thereof, to a volume or filename other than CONSOLE. The first specification is the source file specification and the second is the output file specification

Source file specification consists of a mandatory volume ID, and optional subset-specifying strings, which may be empty. Source file specifications are separated from destination file specifications by a comma (','').

Destination file specification consists of a volume ID, and, if the volume is a block-structured device, a filename.

The most frequent use of this command is to list the entire directory of a volume.

If the directory listed is too long to fit on one screen, the Filer lists as much as it can, and then prompts:

Type <space> to continue

... typing a <space> causes the rest of the directory to be listed; typing an <esc> aborts the listing.

EXAMPLE:

The following display, which represents a complete directory listing for the example disk MYDISK, would be generated by typing any valid volume ID for MYDISK (see Figure 5) in response to the prompt,

Dir listing of what vol?

```
MYDISK:
FILERDOC2.TEXT      38    1-Sep-78
ABSURD.CODE         18    1-Sep-78
HYTYPER.CODE        12    1-Sep-78
STASIS.TEXT          8    1-Sep-78
LETTER1.TEXT        18    1-Sep-78
ASSEMDOC.TEXT       20    1-Sep-78
```

```
FILERDOC1.TEXT     24    1-Sep-78
STASIS.CODE         6    1-Sep-78
10/10 files<listed/in-dir>, 144 blocks used, 376 unused, 200 in largest
```

The bottom line of the display informs the user that 10 files out of 10 files on the disk have been listed, that 144 disk blocks have been used, that 350 disk blocks remain unused, and that the largest area available is 200 blocks.

L(dir transaction involving wildcards:

Prompt: Dir listing of what vol ?

User response: #4:FIL=TEXT

... generates the following display:

```
MYDISK:
FILERDOC2.TEXT     38    1-Sep-78
FILERDOC1.TEXT     24    1-Sep-78
2/10 files<listed/in-dir>, 62 blocks used, 458 unused, 200 in largest
```

EXAMPLE:

L(dir transaction involving writing the directory subset to a device other than CONSOLE:

Prompt: Dir listing of what vol ?

User response: *FIL=TEXT,PRINTER:<return> causes

```
MYDISK:
FILERDOC2.TEXT     38    1-Sep-78
FILERDOC1.TEXT     24    1-Sep-78
2/10 files<listed/in-dir>, 62 blocks used, 458 unused, 200 in largest
```

... to be written to the printer.

EXAMPLE:

L(dir transaction involving writing the directory subset to a block-structured device:

Prompt: Dir listing of what vol ?

User response: #4:FIL=TEXT,#5:TRASH creates the file TRASH on the volume associated with device 5. TRASH would contain:

```
MYDISK:
FILERDOC2.TEXT      38    1-Sep-78
FILERDOC1.TEXT      24    1-Sep-78
 2/10 files<listed/in-dir>, 62 blocks used, 458 unused, 200 in largest
```

III.6.3.9 M(ake

Creates a directory entry with the specified filename.

This command requires the user to type a file specification. Wildcard characters are not allowed. The file size specification option is extremely helpful, since, if it is omitted, the Filer creates the specified file by consuming the largest unused area of the disk. The file size is determined by following the filename with the desired number of blocks, enclosed in square brackets '[' and ']'. Some special cases are:

[0] - equivalent to omitting the size specification.
The file is created in the largest unused area.

[*] - the file is created in the second largest area,
or half the largest area, whichever is larger.

Textfiles must be an even number of blocks, and the smallest possible textfile is four blocks long (two for the header, and two for text). M(ake enforces these restrictions: if the user tries to M(ake a textfile with an odd number of blocks, M(ake will round the number down.

M(ake can be used to create a file (with garbage data) for future use, to extend the size of a file (using the size specification), or to recover a lost file (see Section III.6.4).

EXAMPLE:

Prompt : Make what file?

Response : MYDISK:FARKLE.TEXT[28]

Creates the file FARKLE.TEXT on the volume MYDISK: in the first unused 28-block area encountered.

III.6.3.10 N(ew

Clears the workfile and enables the Editor to create a new workfile.

If there is already a workfile present, the user is prompted:

Throw away current workfile?

Response: 'Y' clears the workfile, while 'N' returns the user to the outer level of the Filer.

If <workfile name>.BACK exists, then the user is prompted:

Remove <workfile name>.BACK ?

Response: 'Y' removes the file in question, while 'N' leaves the .BACK file alone, but does create a new workfile.

A successful N(ew returns the message:

Workfile cleared

III.6.3.11 O(n/off-line

Enables the user to create "subsidiary volumes," or subvolumes, out of existing disk space. Subsidiary volumes can be used in the same ways as the "principal volumes," or physical disk volumes, on which they reside. Refer to Section III.7.2 for more information.

III.6.3.12 P(refix

Changes the current default volume to the <volume name> specified.

This command requires the user to type a volume ID. An entire file specification may be entered, but only the volume ID will be used. It is not necessary for the specified volume to be on-line.

If the user specifies a device number (say, '#5'), then the new default prefix is the name of the volume (e.g., 'CHROME:') in that device. If no volume is in the device when prefix is used, the default prefix remains the device number (e.g., '#5:'), and thereafter, any volume in the default device is the default volume.

To determine the current default volume, the user may respond to the prompt with ':' (see also the V(olume command). To return the prefix to the booted or "Root" volume, user may respond with "*".

If the user invokes the filer P(refix command and enters a device number instead of a disk volume ID and there is no disk on-line in the specified device, the filer will set the system prefix to the device number, rather than to the volume ID of a disk. The system (including the filer) will then consider any disk inserted into the specified device to be the default (prefix) disk. This procedure is exactly equivalent to entering "xp=#n <return>" at the main command line.

CAUTION: When using this facility, the user must remember that the disk in the device is the default disk. It is very easy, in this situation, to assume the system is prefixed to a particular disk, exchange the disks, and write over a valuable file or destroy information.

III.6.3.13 Q(uit

Returns the user to the System (outermost) command level.

R(emove

Reference Manual
File Handling

III.6.3.14 R(emove

Removes file entries from the directory.

This command requires one file specification for each file the user wishes to remove. Wildcards are legal. Size specification information is ignored.

EXAMPLE:

Given the example files (assuming that they are on the default volume):

AARDVARK.TEXT
ANDROID.CODE
QUINCUNX.TEXT
AMAZING.CODE

Prompt: Remove what file?

User Response: AMAZING.CODE

... removes the file AMAZING.CODE from the volume directory.

Note: To remove SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE, the N(ew command should be used, not R(emove, or the System may get confused. Fortunately, before finalizing any removes, the Filer prompts the user with

Prompt: Update directory?

Response: 'Y' causes all specified files to be removed. 'N' returns the user to the outer level of the Filer without any files having been removed.

As noted before, wildcards in R(emove commands are legal.

EXAMPLE:

Prompt: Remove what file?

User Response: A=CODE

... causes the Filer to remove AMAZING.CODE and ANDROID.CODE.

Typing the wildcard '?' causes R(emove to prompt for the removal of each file on a volume. This is useful for 'cleaning out' a directory, and for removing a file which has (inadvertently) been created with a nonprinting character in its name.

Warning: Remember that the Filer considers the file specification '=' (where both subset-specifying strings are empty) to specify every file on the volume. Typing an '=' alone will cause the Filer to remove every file on your directory! (Fortunately, typing 'N' in response to the 'Update directory?' prompt will save your disk from this fate.)

III.6.3.15 S(ave

Saves the workfile under the filename specified by the user.

The entire file specification is not necessary. If the volume ID is not given, the default disk is assumed. Wildcards are not allowed, and the size specification option is ignored.

EXAMPLE:

Prompt: Save as what file?

Response: Type a filename of 10 characters or less. This causes the Filer to automatically remove any old file having the given name, and to save the workfile under that name. For example, typing "X" in response to the prompt causes the workfile to be saved on the default disk as X.TEXT. If a codefile has been compiled since the last update of the workfile, that codefile will be saved as X.CODE.

If a file already exists with the name given, S(ave will respond: 'Destroy old <filename>?'. A 'Y' response causes the old file to be replaced, any other reply exits the S(ave.

The Filer automatically appends the suffixes .TEXT and .CODE to files of the appropriate type. Explicitly typing AFILE.TEXT in response to the prompt will cause the Filer to save this file as AFILE.TEXT.TEXT . Any illegal characters in the filename will be ignored, with the exception of ':'. If the file specification includes a volume id, the Filer assumes that the user wishes to save the workfile on another volume. For example, typing:

RED:EYE

... in response to 'Save as what file?' will generate

MYDISK:SYSTEM.WRK.TEXT -> RED:EYE.TEXT

III.6.3.16 T(ransfer

Copies the specified file or volume to the given destination.

This command requires the user to type two file specifications: one for the source file, and one for the destination file, separated by either a comma or <return>. Wildcards are permitted, and size specification information is recognized for the destination file.

EXAMPLE:

Assume that the user wishes to transfer the file FARKLE.TEXT from the disk MYDISK to the disk BACKUP.

Prompt: Transfer what file ?

User Response: MYDISK:FARKLE.TEXT

Prompt: To where?

Note: On a one-drive machine, do not remove your source disk until you are prompted to insert the destination disk.

User Response: BACKUP:NAME.TEXT

Prompt: Put in BACKUP:
Type <space> to continue

The user should remove the source disk, insert the destination disk and type a <space>. The Filer then notifies the user:

MYDISK:FARKLE.TEXT -> BACKUP:NAME.TEXT

The Filer has made a copy of FARKLE and has written it to the disk BACKUP giving it the name NAME.TEXT. If the specified file is large, the user may be prompted to alternately insert the source and destination disks until the transfer is completed.

It is often convenient to transfer a file without changing the name, and without retyping the filename. The Filer enables the user to do this by allowing the character '$' to replace the filename in the destination file specification. In the above example, had the user wished to save the file FARKLE.TEXT on BACKUP under the name FARKLE.TEXT, she could have typed:

MYDISK:FARKLE.TEXT,BACKUP:$

Warning: Avoid typing the second file specification with the filename completely omitted! For example, a response to the Transfer prompt of the form:

MYDISK:FARKLE.TEXT,BACKUP:

generates the message:

Destroy BACKUP: ?

... a 'Y' answer causes the directory of BACKUP to be wiped out! See Section III.6.4 for a way to recover.

Also note: If the file you are T(ransfer'ring is two blocks long or less, you will not even receive the warning prompt.

Files may be transferred to volumes that are not block structured, such as CONSOLE: and PRINTER:, by specifying the appropriate volume ID (see Figure 5) in the destination file specification. A filename on a non-block-structured device is ignored. It is generally a good idea to make certain beforehand that the destination volume is on-line.

EXAMPLE:

Prompt: Transfer what file?

User Response: FARKLE.TEXT

Prompt: To where?

User Response: PRINTER:

... causes FARKLE.TEXT to be written to the printer.

The user may also transfer from non-block-structured devices, provided they are input devices (the source file must end with an <eof> (ASCII ETX) or the Filer will not know when to stop transferring!). Filenames accompanying a non-block-structured device ID are ignored.

The wildcard capability is allowed for T(ransfer. If the source file specification contains a wildcard character, and the destination file specification involves a block-structured device, then the destination file specification must also contain a wildcard character. The subset-specifying strings in the source file specification will be replaced by the analogous strings in the destination file specification (henceforward known as replacement strings). Any of the subset-specifying or replacement strings may be empty. Remember that the Filer considers the file specification '=' to specify every file on the volume.

EXAMPLE:

Given the volume MYDISK containing the files PAUCITY, PARITY and PENALTY, and the destination ODDNAMZ:

Prompt: Transfer what file?

User Response: P=TY,ODDNAMZ:V=S

would cause the Filer to reply:

```
MYDISK:PAUCITY          -> ODDNAMZ:VAUCIS
MYDISK:PARITY           -> ODDNAMZ:VARIS
MYDISK:PENALTY          -> ODDNAMZ:VENALS
```

Using '=' as the source filename specification will cause the Filer to attempt to transfer every file on the disk. This will probably overflow the output buffer. (There are easier ways to transfer whole disks. If you wish to do this, please refer to the material in this section on volume-to-volume transfers.)

Using '=' as the destination filename specification will have the effect of replacing the subset-specifying strings in the source specification with nothing. A brief reminder: '?' may be used in place of '='. The only difference is that '?' causes the user to be asked for verification before the operation is performed.

A file can be transferred from a volume to the same volume by specifying the same volume ID for both source and destination file specifications. This is frequently useful when the user wishes to relocate a file on the disk. Specifying the number of blocks desired will cause the Filer to copy the file in the first-fit area of at least that size. If no size specification is given, the file is written in the largest unused area.

If the user specifies the same filename for both source and destination on a same-disk transfer, then the Filer rewrites the file to the size-specified area, and removes the older copy.

EXAMPLE:

Prompt: Transfer what file?

User Response: #4:QUIZZES.TEXT,#4:QUIZZES.TEXT[20]

... causes the Filer to rewrite QUIZZES.TEXT in the first 20-block area encountered (counting from block 0) and to remove the previous version of QUIZZES.TEXT.

It is also possible to do entire volume-to-volume transfers. The file specifications for both source and destination should consist of volume ID only. Transferring a block-structured volume to another block-structured volume causes the destination volume to be 'wiped out' so that it becomes an exact copy (including directory) of the source volume.

Note that some disks have areas which are not accessible by the System. Those areas cannot be transferred by the Filer. Bootstraps, in particular, may have to be transferred with the utility BOOTER. See the Introduction to the UCSD p-System for more details.

EXAMPLE:

Assume that the user desires an extra copy of the disk MYDISK: and is willing to sacrifice disk EXTRA:

Prompt: Transfer what file?

User Response: MYDISK:,EXTRA:

Prompt: Destroy EXTRA: ?

Warning: If the user types 'Y', the directory of EXTRA: is destroyed! An 'N' response returns the user to the outer level of the Filer, and a 'Y' causes EXTRA to become an exact copy of MYDISK. Often this is desirable for backup purposes, since it is relatively easy to copy a disk this way, and the volume name can be changed (see C(hng) if desired.

Although it is possible to transfer a volume (disk) to another using a single disk drive, it is a tedious process, since the transfer in main memory reads the information in rather small chunks, and a great deal of disk juggling is necessary for the complete transfer to take place.

V(olumes

III.6.3.17 V(olumes

Lists volumes currently on-line, with their associated volume (device) numbers and sizes.

A typical display might be:

```
Vols on-line
   1    CONSOLE:
   2    SYSTERM:
   4 #  FLOPPY1: [520]
   5 #  FLOPPY2: [520]
   6    PRINTER:
  11 #  WNCHSTR: [2235]
Root vol is - FLOPPY1:
Prefix is   - FLOPPY2:
```

The system volume ('Root vol') is the default volume unless the prefix (see P(refix) has been changed. Block-structured devices are indicated by '#'.

After each disk volume, the number of 512-byte blocks that it contains is given in square brackets. This can be useful if the system uses disks of varying storage capacities. In the preceding example, the Winchester Disk on-line in drive #11: contains 2235 blocks of storage capacity, and the floppies on-line in drives #4: and #5: contain 520 blocks.

The V(olumes command also displays the mounted subsidiary volumes. The name of the principal volume and the name of the starting block are given for each subsidiary volume listed.

III.6.3.18 W(hat

Identifies the name and state (saved or not) of the workfile.

EXAMPLE:

```
Workfile is DOC1:STUFF
```

III.6.3.19 eX(amine

Attempts to physically recover suspected bad blocks.

The user must specify the name of a volume that is on-line.

EXAMPLE:

Prompt : Examine blocks on what volume?

Response : <volume ID> generates the

Prompt: Block-range ?

The user should have just done a bad block scan, and should enter the block number(s) returned by the bad block scan. If any files are endangered, the following prompt should appear:

Prompt: File(s) endangered:
<filename>
Fix them?

Response: 'Y' will cause the Filer to examine the blocks and return either of the messages:

Block <block-number> may be ok

... in which case the bad block has probably been fixed, or

Block <block-number> is bad

... in which case the Filer will offer the user the option of marking the block(s) BAD. Blocks which are marked BAD will not be shifted during a K(runch, and will be rendered unavailable and effectively harmless (though they do reduce the amount of room on your disk).

An 'N' response to the 'fix them?' prompt returns the user to the outer level of the Filer.

Warning: A block which is 'fixed' may contain garbage. 'May be ok' should be translated as 'is probably physically ok'. Fixing a block means that the block is read, is written back out to the block and is read again. If the two reads are the same, the message is 'may be ok'. In the event that the reads are different, the block is declared bad and may be marked as such if so desired.

III.6.3.20 Z(ero

Sets up an empty directory on the specified volume. The previous directory is rendered irretrievable.

EXAMPLE:

Prompt: Zero dir of what vol ?

Response: <volume ID>

Prompt: Destroy <volume name> ?

Response: A 'Y' response generates ...

Prompt: Duplicate dir ?

Response: If a 'Y' is typed, then a duplicate directory will be maintained. This is advisable because, in the event that the disk directory is destroyed, a utility program called COPYDUPDIR can use the duplicate directory to restore the disk.

The following two prompts only appear if there was a directory on the disk before the Z(ero command was used:

Prompt: Are there 520 blks on the disk ? (y/n)

Response: 'N' generates ...

Prompt: # of blocks on the disk ?

(... which also appears if the disk was blank.)

Response: User types the number of blocks desired. This number varies depending on the hardware used.

'Y' generates ...

Prompt: New vol name ?

Response: User types any valid volume name.

Prompt: <new volume name> correct ?

Response: 'Y' causes the Filer, if it succeeds in writing the new directory on the disk, to respond with the message:

<new volume name> zeroed

Z(ero writes a directory of the same byte sex as the processor that is running Filer. This is true even if the disk had a prior directory of opposite byte sex. All other Filer commands leave the directory's byte sex unchanged. More information on byte sex may be found in the Installation Guide.

III.6.4 Recovering Lost Files

Sometimes a file is removed by accident, or its directory entry is written over for one reason or another. While the initial response of the user is typically a scream, there are often ways to recover the information that has apparently been lost. This section outlines some of the ways to recover files that have been lost, and also describes what may be done if the user loses an entire directory.

When a file is removed, it is still on disk, but no longer in the directory. The information that it contained remains there until another file is written over it (which could happen at any time, since the Filer considers it usable space). If a file is accidentally removed, the user must be careful not to perform any actions (whether from the System or from a user program) that write to the disk, since it is possible they will overwrite the lost file. The K(runch command is virtually guaranteed to do this: avoid it.

The E(xtended list command in the Filer will display both files in the directory and <UNUSED> blocks that have once contained files. Usually, by looking at the length of an unused portion and its location in the directory, the user will be able to tell where the lost file is; using the M(ake command to re-create a file in the same location will recover the lost file.

To recover a lost file with M(ake, the size specification should be equal to the size of the file that was lost. If the user remembers this, or if the lost file was adjacent on both sides to files that are still listed in the directory, this presents no difficulty. If the user does not remember where the file was or how large it was, see below.

Since M(ake makes a file of the specified size in the first available location, it may be necessary to M(ake dummy files that fill up unused (and unwanted) space which precedes the location of the file that was lost. These dummy files may later be removed.

EXAMPLE:

```
WORK:
SYSTEM.MISCINFO      1   5-Aug-80      6   512  Datafile
< UNUSED >           1                 7
SYSTEM.SYNTAX       14   5-Aug-80      8   512  Datafile
REM.WRK.CODE         4   5-Aug-80     22   512  Codefile
< UNUSED >          75                26
MYFILE.TEXT         20   9-Nov-80    101   512  Textfile
< UNUSED >         373               121
 4/4 files<listed/in-dir>, 45 blocks used, 475 unused, 373 in largest
```

If MYFILE.CODE was 4 blocks long and used to be located just after MYFILE.TEXT, it can be re-created by M(aking FILLER[75] in order to fill up the 75-block unused space on the disk. Next, M(ake MYFILE.CODE[4]. MYFILE.CODE will again be located immediately following MYFILE.TEXT. Finally, R(emove FILLER from the directory. The resulting E(xtended directory listing is:

```
WORK:
SYSTEM.MISCINFO     1   5-Aug-80      6    512  Datafile
< UNUSED >          1                 7
SYSTEM.SYNTAX      14   5-Aug-80      8    512  Datafile
REM.WRK.CODE        4   5-Aug-80     22    512  Codefile
< UNUSED >         75                26
MYFILE.TEXT        20   9-Nov-80    101    512  Textfile
MYFILE.CODE         4   9-Nov-80    121    512  Codefile
< UNUSED >        369               125
5/5 files<listed/in-dir>, 49 blocks used, 471 unused, 369 in largest
```

One further note: in order to eX(ecute a codefile, it is necessary that the codefile was created with a .CODE suffix; the codefile's name may later be changed. If the user has lost a codefile that does not have a .CODE suffix (for example, SYSTEM.FILER), the M(ake command should remake the file with the .CODE suffix (e.g., FILER.CODE), and then change the name back (e.g., to SYSTEM.FILER again). If this is not done, it will be impossible to eX(ecute the re-created file.

If the user cannot determine or remember where the file was located on the disk the RECOVER program should be used. RECOVER will scan the directory for entries which look valid. If that search does not yield the desired file, it will attempt to read the entire disk looking for areas which resemble files, and ask the user if it should attempt to re-create them. RECOVER is described in detail in Section IX.7 of this document.

If RECOVER fails to find desired information, the user's last resort is to use the PATCH utility to manually search through the disk. Once data has been found, a Pascal program may be written to read data with the UNITREAD or BLOCKREAD intrinsics, and write it to a newly created file.

If a directory entry seems erroneous or inexplicable, the PATCH utility may be used to examine the exact contents of the directory. Similarly, if the user desires to examine a particular block on a disk to determine whether it is part of a lost file, the PATCH utility may be used. A detailed description of PATCH appears in Chapter IX. The following paragraph outlines the use of PATCH in this particular context; the reader should also refer to Section IX.2.

In order to look at a directory using PATCH, the user should eX(ecute PATCH, then type a <return> in response to PATCH's G(et command. PATCH then prompts for the device number of the disk in question. Answer this prompt, then R(ead block 2: this is the beginning of the directory. In order to see the directory printed out in characters (as opposed to hex, octal, or decimal digits, which in this context are not very useful!), type M(ix V(iew. In order to examine the remainder of the directory, use the F(orward command. The directory spans blocks 2, 3, 4, and 5. If a duplicate directory is present, it occupies blocks 6..9.

Examining other blocks of a disk is a routine use of PATCH, described fully in Section IX.2.

III.6.4.1 Lost Directories

Losing a disk directory can prove even more frustrating than losing a single file. If there were no software tools for doing so, many hours could be spent trying to recreate a lost disk. This section describes some of the things that can be done, and should help ease the pain of re-creating a directory that has been lost.

Recovering a disk is simplest when the disk contains a duplicate directory. The directory spans blocks 2..5 on a disk. If a duplicate directory is present, it spans blocks 6..9. Every time the directory is altered, the duplicate directory is updated as well, thus providing a convenient backup.

If a directory is lost on a disk that has a duplicate directory, the COPYDUPDIR utility may be used to simply move the duplicate directory to the location of the standard disk directory. This should be all the recovery that is necessary.

If after reading this you decide to put duplicate directories on all of your disks, there are two methods available. The first is to use the Z(ero command when first creating a disk with the F(iler. When the prompt 'Duplicate dir?' appears, answer 'Y' for yes.

If a disk is already in use and contains only one directory, the utility MARKDUPDIR will create a duplicate directory. However, caution must be exercised when using this utility: blocks 6..9 of the disk (the location of the duplicate directory) must be unused, or file information will be lost.

The use of COPYDUPDIR and MARKDUPDIR is fully described in Section IX.4.

In the unhappy event that a directory is lost, and no duplicate directory was present, the user should use the RECOVER utility already mentioned. RECOVER is described in Section IX.7.

If the Filer E(xtended list or L(ist commands are used, and specify an optional output file, and the filename given for the output file is a disk volume without a filename, the directory is destroyed.

EXAMPLE:

L(ist directory will prompt:

Dir listing of what vol ?

Response: MYDISK:, MYDISK: <return>
Response: MYDISK:,: <return>

... either of these responses cause the first few blocks (aproximately 6) of MYDISK: to be overwritten with a listing of the directory of MYDISK:.

Response: MYDISK:, DISK2:

... causes the directory of DISK2: to be overwritten.

In the latter case, the disk recovery methods already described must be used. In the first two cases, recovery is not so difficult, even if there was no duplicate directory, since MYDISK:'s directory has been overwritten with what is essentially a copy of itself.

First, get a copy of the directory listing of MYDISK:. (If MYDISK: was your System disk, you will have to boot another System.) Use the Filer to T(ransfer 'MYDISK:' to an output device: PRINTER:, REMOUT:, or CONSOLE:.

Once you have a hard copy of the directory (if you transferred to CONSOLE:, write it down!), use the Filer to Z(ero MYDISK:. The Z(ero command will not alter the contents of MYDISK:, only the directory itself. Now use the M(ake command to remake all of the files on the disk (as described above).

III.7 Subsidiary Volumes

The purpose of subsidiary volumes is to provide two levels of directory hierarchy and to expand the p-System's ability to use large storage devices such as Winchester disk drives. Currently, p-System disk volumes contain a 4-block directory located in blocks 2 through 5. The rest of the disk is occupied by the actual files described in the directory. The size of the directory allows for a maximum of 77 files to reside on the corresponding disk image.

Subsidiary volumes are virtual disk images that actually reside within a standard p-System file. (The physical disk is referred to as the principal volume.) Each subsidiary volume may contain up to 77 files.

A subsidiary volume appears in the directory of the principal volume as a file. Subsidiary volume file names can have a maximum of seven characters and must be followed by the suffix ".SVOL". The following listing is an example.

```
MAIL.SVOL
TESTS.1.SVOL
DOC_B.SVOL
```

The subsidiary volume disk image resides within the actual .SVOL file. The directory format and file formats are the same as for any other p-System disk volume. The volume ID of the subsidiary volume is that portion of the corresponding file name that precedes the ".SVOL". For example, the three preceding files would contain the following subsidiary volumes.

```
MAIL:
TESTS.1:
DOC_B:
```

III.7.1 Creating and Accessing Subsidiary Volumes

To create a subsidiary volume, use the filer M(ake command and the file name suffix, .SVOL. As with any other file created by the M(ake command, the subsidiary volume will occupy:

1. All of the largest contiguous disk area if created as follows:

   ```
   Make what file? DOCS.SVOL
   ```

2. Half of the largest area or all of the second largest area, whichever is larger, if created as follows:

   ```
   Make what file? DOCS.SVOL[*]
   ```

3. A specified number of blocks, in the first area large enough to hold that many blocks, if created as in the following examples:

```
Make what file? DOCS.SVOL[200]
Make what file? DOCS.SVOL[1500]
```

After you have entered the M(ake command to create a subsidiary volume, the filer will display the following prompt.

```
Zero subsidiary volume directory?
```

If the user responds with a "Y", the directory of the new subsidiary volume will be zeroed. If the user types an "N", the directory will not be zeroed, and any files that may have existed on a previous subsidiary volume in the same location will reappear within the directory. In both cases, the number of blocks indicated within the directory will always correspond to the size of the actual .SVOL file.

Subsidiary volumes may not be nested. That is, do not M(ake a .SVOL file within another .SVOL file.

When a subsidiary volume is created, it is automatically mounted and may be accessed and used like any other p-System volume. The filer command V(olumes will indicate that the new volume is on-line, and will show its corresponding device number (for example, #24:). Either the volume ID or the device number may be used when referencing the subsidiary volume. Files may now be placed on the new subsidiary volume, and all of the applicable file commands may reference it.

III.7.2 Mounting and Dismounting Subsidiary Volumes

This section describes how to mount and dismount subsidiary volumes. It should be noted that a mounted subsidiary volume is subtly different from an on-line subsidiary volume.

A mounted subsidiary volume means that the p-System is aware of the existence of the volume and sets aside a device number for it (e.g., #24:) It is required that a subsidiary volume be mounted before it can be used. While it is mounted, only that specific subsidiary volume will correspond to that device number. A subsidiary volume stays mounted until it is dismounted. Once mounted, it is on-line anytime its principal volume is in the disk drive. It is off-line when the principal volume has been removed from the disk drive.

CAUTION: There is a danger of confusing the p-System if two principal volumes each contain a subsidiary volume in the same location with the same name. This might easily be the case where backup disks are used. If these principal volumes are swapped in and out of the same drive, and the similar subsidiary volumes are

accessed, the filer may become confused in the same way that it can when any two on-line volumes have the same name.

CAUTION: Low level I/O routines, like UNITWRITE, must be used cautiously with subsidiary volumes and principal volumes. If a principal volume is removed from a disk drive, and another disk is inserted, these low level routines have no way of knowing that the subsidiary volumes that were mounted on the original disk are no longer present. Doing a UNITWRITE to absent subsidiary volumes under these circumstances will overwrite data on the disk presently occupying the disk drive.

When the p-System is booted, all of the on-line disks are searched for .SVOL files. All of the corresponding subsidiary volumes are then mounted. The same process occurs whenever the p-System is I(nitialized.

The booting or initializing process will mount as many subsidiary volumes as it finds as long as there is room in the p-System unit table. If the unit table becomes full, no more subsidiary volumes will be mounted with no warning given.

If, after booting or initializing, the user places a new physical disk on-line, any subsidiary volumes contained on it must be manually mounted if they are to be accessed.

In order to mount or dismount subsidiary volumes, the filer has a new command: O(nline/offline. From the main filer prompt type "O" and the following will appear:

Subsidiary Volume: M(ount, D(ismount

To mount a subsidiary volume, the user can type "M" and receive the following promp.

Mount what vol ?

To dismount a subsidiary volume, the user can type "D" and receive the following prompt.

Dismount what vol ?

Suppose that a principal volume, P_VOL:, contains the following files:

```
P_VOL:
  FILE1.TEXT
  FILE1.CODE
  VOL1.SVOL
  FILE2.TEXT
  FILE2.CODE
  DOC1.SVOL
  -FUN-.SVOL
```

To mount subsidiary volumes on P_VOL:, the user can respond to the mount prompt with the file name as in the following examples.

```
Mount what vol ? VOL1.SVOL<return>
Mount what vol ? VOL1.SVOL,-FUN-.SVOL<return>
Mount what vol ? P_VOL:=<return>
Mount what vol ? #5:=<return>
```

The first example mounts VOL1:, the second mounts VOL1: and -FUN-:, the third mounts all three subsidiary volumes on P_VOL:, and the fourth example mounts all subsidiary volumes on the disk in drive #5:.

To dismount any of these volumes, the user can respond to the dismount prompt with the VOLUME ID as in the following examples:

```
Dismount what vol ? #24:
Dismount what vol ? VOL1:<return>
Dismount what vol ? VOL1:, DOC1:, -FUN-:<return>
```

The first example dismounts the subsidiary volume associated with the device number 24. The second example dismounts VOL1:, and the third example dismounts three subsidiary volumes.

Note that the SYSTEM.MISCINFO field "MAX NUMBER OF SUBSIDIARY VOLS" is initially set to support up to 10 subsidiary volumes.

III.8 User-Defined Serial Devices

The user may have up to three serial devices in addition to the standard CONSOLE:, REMIN:, and REMOUT: devices. User-defined serial devices may include additional printer ports, additional consoles, communication lines between users in a multi-user environment, etc. Refer to appendix D for more information.

IV. THE SCREEN ORIENTED EDITOR

IV.0 Introduction

This introduction describes the general environment of using the Screen Oriented Editor (called the Editor throughout this chapter). Section IV.1 is a tutorial for the novice, Section IV.2 describes general conventions of the Editor, and Section IV.3 contains a detailed description of each command, with examples, in alphabetical order.

IV.0.1 The Concept of a Window into the File

The Screen Oriented Editor is specifically designed for use with Video Display Terminals (or Cathode Ray Tubes -- CRTs). On entering any file, the Editor displays the start of the file on the second line of the screen. If the file is too long for the screen, only the first lines of it are displayed. Most screens have 24 lines, and in the Editor one line is used for the prompts; thus, the Editor typically displays only 23 lines of a file at any one time. This is the concept of a "window." The whole file is there and is accessible through Editor commands, but only a portion of it can be seen through the "window" of the screen. When any Editor command takes the user to a position in the file which is not already displayed, the window is updated to show that new portion of the file.

IV.0.2 The Cursor

The cursor represents the user's exact position in the file, and can be moved to any position. The window shows the portion of the file that surrounds the cursor; to see another portion of the file, move the cursor. Action always takes place at the cursor. Some of the commands permit additions, changes or deletions of such length that the screen cannot hold the whole portion of the text that has been changed. In these cases, the portion of the screen where the cursor finally stops is displayed. In no case is it necessary for the user to operate on portions of the text not seen on the screen, but in some cases it is optional.

In this chapter, all text examples are shown in upper case, and the cursor is denoted by an underline or a lower case character.

IV.0.3 The Promptline

The Editor displays a promptline to remind the user of the current command and the options available for that command. Only the most commonly used options appear on the promptline. The promptline is always displayed at the top of the screen, and the Editor's outer promptline looks like this:

>Edit: A(djust C(opy D(el F(ind I(nsert J(ump K(ol M(argin P(age ? [IV.1 F6c]

IV.0.4 Notation Conventions

The notation used in this chapter corresponds to the notation used by the Editor to prompt the user. Any input that is enclosed between '<' and '>' is requesting that a particular key be used, not that the particular word be typed out. For example, <RET> or <return> means that the return key should be typed at that point. When a particular sequence of key strokes is required they will be contained within quotes. For example, 'FILENAME'<return> represents typing the sequence 'FILENAME' and then typing the return key. Either lower or upper case may be used when typing Editor commands.

IV.0.5 The Editing Environment Options

The Editor has two chief environments: one for entering and modifying programs, and one for entering and modifying English (or language of your choice) text. The first mode includes automatic indentation and search for isolated tokens, the second mode includes automatic text filling. For more detail on these two options, see the description below of the E(nvironment option of the S(et command.

IV.1 Getting Started

The Editor is designed to handle any textfiles, whether programs, data, or documents. This tutorial section uses a sample program to illustrate the use of the most basic Editor commands. You may find it easier to follow if you have the System running in front of you, and can duplicate the examples on your own.

IV.1.1 Entering the Workfile and Getting a Program.

When you enter the Editor, the text of the workfile is read and displayed. If you have not already created a workfile, then this prompt appears:

No workfile is present. File? (<ret> for no file <esc> to exit)
:

There are three ways to answer this question :

1) With a name, for example 'STRING1'<ret>. The file named STRING1.TEXT will now be retrieved. The file STRING1 could contain a program, also called STRING1, as in Figure IV.1. After typing the name, a copy of the text of the first part of the file appears on the screen.

Figure IV.1

```
PROGRAM STRING1;
BEGIN
  WRITE('TOO WISE');
  WRITE('YOU ARE');
  WRITELN(',');
  WRITELN('TOO WISE');
  WRITELN('YOU BE')
END.
```

2) With a <return>. This implies that a new file is to be started. The only thing visible on the screen after doing this is the Editor promptline. A new workfile is opened and currently has nothing in it. Type 'I' to begin inserting a program or text.

3) With <escape>. This causes the Editor to quit and return you to the System command level. Useful when you didn't mean to type 'E'.

IV.1.2 Moving the Cursor

In order to edit, it is necessary to move the cursor. On the keyboard are four keys with arrows: these move the cursor. The <up-arrow> moves the cursor up one line, the <right-arrow> moves the cursor right one space, and so forth.

The cursor cannot be moved outside the text of the program. For example, after the 'N' in 'BEGIN' in Figure IV.2, push the <right-arrow> and the cursor will move to the 'W' in 'WRITE'. Similarly, at the 'W' in "WRITE('TOO WISE ');", use <left-arrow> to move to after the 'N' in 'BEGIN'.

Figure IV.2

```
BEGIN
  WRITE('TOO WISE ');

BEGIN
  wRITE('TOO WISE ');
```

If it is necessary to change the "WRITE('TOO WISE ');" found in the third line to a "WRITE('TOO SMART ');", the cursor must first be moved to the right spot.

For example: if the cursor is at the 'P' in 'PROGRAM STRING1;', go down two lines by pressing the down arrow twice. To mark the positions the cursor occupies, labels a,b,c are used in Figure IV.3. 'a' is the initial position of the cursor; 'b' is where the cursor is after the first <down-arrow>; 'c', after the second <down-arrow>.

Figure IV.3

```
aROGRAM STRING1
bEGIN
c WRITE('TOO WISE ');
```

Now, using the <right-arrow>, move the cursor until it sits on the 'W' of 'WISE'. Note that with the use of <down-arrow> the cursor appears to be outside the text (c). Actually it is at the 'W' in 'WRITE', so do not be surprised when on typing

the first <left-arrow> the cursor jumps to the 'R' in 'WRITE'. The point is that when the cursor is displayed outside the text, it is conceptually on the closest character to the right or left.

IV.1.3 Using Insert

The Editor promptline shows that you may I(nsrt (insert) text by typing 'I'. The cursor must be in the correct position before typing 'I'. Earlier, the cursor was moved to the 'W' in 'TOO WISE'; now, on typing 'I', an insertion will be made before the 'W'. The rest of the line from the point of insertion will be moved to the right hand side of the screen. If the insertion is lengthy, that part of the line will be moved down to allow room on the screen. After typing 'I' the following promptline will appear on the screen:

>Insert: Text {<bs> a char, <del> a line} [<etx> accepts, <esc> escapes]

If this promptline does not appear at the top of the screen, then you may have accidentally typed some character other than 'I'.

If the cursor is at the 'W' in 'WISE', and typing 'I' causes the insert promptline to appear, 'SMART' may be inserted by typing those five letters. They will appear on the screen as they are typed.

There remains one more important step. The choice at the end of the prompt line indicates that pushing the <etx> key accepts the insertion, while pushing the <esc> key rejects the insertion and the text remains as it was before typing 'I'.

Figure IV.4 (Screen after typing 'SMART')

```
BEGIN WRITE('TOO SMART                                                  WISE ');
```

Figure IV.5 (Screen after <etx>)

```
BEGIN
  WRITE('TOO SMARTWISE ');
```

Figure IV.6 (Screen after <esc>)

```
BEGIN
  WRITE('TOO WISE ');
```

It is possible, and indeed often necessary, to insert a carriage return. This is done by typing <return> while in I(nsert. This causes the Editor to start a new line. Notice that a carriage return starts a new line with the same indentation as the previous one. This is intended as a programming aid.

IV.1.4 Using Delete

D(elete works like I(nsert. Having inserted 'SMART' into the STRING1 program and having pushed <etx>, 'WISE' must be deleted. Move the cursor to the first of the items to delete and type 'D' to use the D(elete command. The following promptline should appear:

>Delete: <> <moving commands> {<etx> to delete, <esc> to abort}

Each time <space> is typed a letter disappears. In this example typing 4 spaces will cause 'WISE' to disappear. The <backspace> character will undo the deletion one character at a time. Now the same choice must be made as in I(nsert. Type <etx> and the proposed deletion is made or type <esc> and the proposed deletion reappears and remains part of the text.

It is possible to delete a carriage return. At the end of the line, enter D(elete, and <space> until the cursor moves to the beginning of the next line.

These commands alone are sufficient to edit any file desired. The next section describes many more commands in the Editor which make editing much easier.

IV.1.5 Leaving the Editor and Updating the Workfile

When all the changes and additions have been made, exit the Editor and save a copy of the modified program. This is done by typing 'Q' which will cause the prompt shown in Figure IV.7.

Figure IV.7

```
>Quit:
     U(pdate the workfile and leave
     E(xit without updating
     R(eturn to the editor without updating
     W(rite to a file name and return
```

The most elementary way to save a copy of the modified file on disk is to type 'U' for U(pdate which causes the workfile to be saved as SYSTEM.WRK.TEXT. With the workfile thus saved, it is possible to use the R(un command, provided of course the file is a program. It is also possible to use the S(ave option in the Filer to save the modified workfile under a different name before using the Editor to modify or create another file.

Section IV.3.10 explains in greater detail the options available at Q(uit.

IV.2 Using the Editor

IV.2.1 Command Hierarchy

Some commands in the Editor perform a function directly, but most constitute a "second level" of command. These are commands which display a promptline of their own, with another set of commands you may use. All of these subordinate commands, even Q(uit, allow you to return to the outer Editor level, either after performing some special function, or without having affected anything (possibly as an escape from accidentally invoking the command).

Each description of a second level command includes a sample of the secondary promptline. Some commands, like S(et, even have third level prompts. In all cases, you may move both down the command tree and up it, returning to the "root" of the outer Editor prompt. This root is itself just one branch of the System command tree, as pictured in Chapter I.

This is a possibly too-wordy description of a concept which is very easy to visualize if you are sitting at a terminal and using the Editor.

IV.2.2 Repeat Factors

Most of the commands allow repeat factors. A repeat factor is applied to a command by typing a number immediately before the command's letter. The command is then repeated for the number of times indicated by the repeat factor. For example: typing '2'<down-arrow> will cause the <down-arrow> commmand to be executed twice, moving the cursor down two lines. Commands which allow a repeat factor assume the repeat factor to be 1 if no number is typed before the command. A '/' can be used as a repeat factor, and means repeat the command until the end (or beginning) of the textfile is encountered.

IV.2.3 The Cursor

The cursor is displayed "on top of" a character, but it is conceptually in front of that character. In other words, the cursor is never "at" a character, but always between two characters. This is a convention which you must remember in order to use the I(nsert and D(elete commands.

IV.2.4 Direction

There is a global direction for all commands in the Editor. It affects certain commands, and certain methods of cursor movement. This direction is indicated by the first character in the promptline: either > or <, for forward and backward, respectively. The direction can be changed by the characters indicated in the next section below.

When the Editor is first entered, the global direction is forward.

IV.2.5 Moving the Cursor

The Cursor can be moved by a number of means. One obvious method is to use the cursor keys.

Another method is to use traditional typewriter characters, i.e., <space bar>, <return>, <tab>, and <backspace>. The former three are affected by the global direction. The arrow keys and <backspace> are not.

Typing an '=' causes the cursor to jump to the beginning of the last section of text which was inserted, found, or replaced, and sets the 'equals mark' to the cursor's location. Equals works from anywhere in the file and is not affected by the global direction. An I(nsert, F(ind, or R(eplace causes the position (within the workfile) of the beginning of the insertion, find, or replacement to be saved. Typing '=' causes the cursor to jump to that position, and saves the cursor location. If a C(opy or a D(elete has been made between the beginning of the file and that absolute position, the cursor will not jump to the start of the insertion, as that absolute position will have been lost.

Two alphabetic commands are meant explicitly for moving the cursor. J(ump will move it to the beginning or end of the file, or to a marker which the user has previously defined. P(age moves the window forward (or backward) one screenful, and positions the cursor at the beginning of the line. Refer below to the full descriptions of these commands.

A variety of other commands reposition the cursor in addition to performing their specific actions. Thus, A(djust moves the cursor along with the entire line, C(opy

and I(nsert move the cursor to the end of their insertions, F(ind and R(eplace leave the cursor after their last successful hit, and V(erify places the cursor in the middle of the screen. Full details of all these actions are found below.

The following is a summary of cursor-moving characters:

Not sensitive to the current global direction:

<down-arrow>	Moves cursor down
<up-arrow>	Moves cursor up
<right-arrow>	Moves cursor right
<left-arrow>	Moves cursor left
<backspace>	Moves cursor left
'<' or ',' or '-'	Changes the global direction to backward
'>' or '.' or '+'	Changes the global direction to forward

Sensitive to the global direction:

<space>	Moves cursor one space in the global direction
<tab>	Moves cursor to the next tab stop; tab stops are every 8 spaces, starting at the left of the screen
<return>	Moves cursor to the beginning of the next line

Note: The period and the comma can also be used to change direction because on many standard keyboards, '.' is lower-case for '>' and ',' is lower-case for '<'.

Repeat factors can be used with any of the above commands.

For user convenience, the Editor maintains the column position of the cursor when using <up-arrow> and <down-arrow>. When the cursor is outside the text, the Editor treats the cursor as though it were immediately after the last character, or before the first, in the line.

IV.2.6 Entering Strings in F(ind and R(eplace

Both F(ind and R(eplace operate on delimited strings. The Editor has two string storage variables. One, called <targ> by the promptlines, is the target string and is used by both commands, while the other, called <sub> by R(eplace's promptline, is the substitute string and is used only by R(eplace.

These strings are entered when you use F(ind or R(eplace. Once entered, they are saved by the Editor and may be re-used.

When you enter a string, it must be delimited by two occurrences of the same character. For example, '/fun/', '$work$', and '"gismet"' represent the strings 'fun', 'work', and 'gismet', respectively. The Editor allows any character which is not a letter or a number to be used as a delimiter.

There are two search modes -- Literal and Token. These modes are stored by the S(et E(nvironment command, and can be changed by it (see below), or they may be temporarily overriden when you use F(ind or R(eplace (refer to descriptions of these commands).

In Literal mode, the Editor looks for any occurrences of the target string. In Token mode the Editor looks for isolated occurrences of the target string. The Editor considers a string isolated if it is surrounded by spaces or other punctuation. For example, in the sentence 'Put the book in the bookcase.', using the target string 'book', Literal mode will find two occurrences of 'book' while Token mode will find only one -- the word 'book', isolated by <space><space>.

In addition, Token mode ignores spaces within strings, so that both '(",")' and '(",")' are considered to be the same string.

When using either F(ind or R(eplace, you may use the strings you have previously entered by typing 'S'. For example, typing 'RS/'<any-string>'/' causes the R(eplace mode to replace the previous target string, while typing 'R/'<any-string>'/S' causes the target string to be replaced with the previous substitute string.

To see what the <targ> and <sub> strings are at any given time, use the S(et E(nvironment command.

More specific information on this topic is given below under the descriptions of F(ind, R(eplace, and S(et E(nvironment.

IV.3 Screen Oriented Editor Commands

Each command (and its sub-commands, if any) is fully described below. Commands are listed in alphabetical order, and the descriptions, which include examples, are meant to be used both for reading and for reference.

IV.3.1 A(djust

On the promptline: A(djust.

Repeat factors are allowed.

A(djust displays the following prompt:

>Adjust: L(just R(just C(enter <arrow keys> {<etx> to leave}

A(djust is used to adjust indentation. The <right-arrow> and <left-arrow> commands move the line on which the cursor is located. Each time a <right-arrow> is typed the whole line moves one space to the right. Each <left-arrow> moves it one space to the left.

To adjust a whole sequence of lines, adjust one line, then use <up-arrow> (or <down-arrow>) commands and the line above (below) will be automatically adjusted by the same amount.

This feature can be used to align a whole set of lines. If you adjust a line horizontally, then using <up-arrow> (or <down-arrow>) now causes the line above (below) to be adjusted by the sum of previous adjustments. In other words, the horizontal offset accumulates until A(djust is exited with <etx>.

The character 'L' justifies the line to the left margin, 'R' justifies it to the right margin, and 'C' centers the line between the margins. <up-arrow>s and <down-arrows> can be used to duplicate the adjustment on preceding (succeeding) lines, as above.

The margins can be altered with the S(et E(nvironment command. See Section IV.3.12.2.

The cursor is repositioned at the beginning of the last line adjusted. <etx> is the only way to exit the A(djust command; <esc> will not work.

IV.3.2 C(opy

On the promptline: C(opy.

Repeat factors not allowed.

C(opy displays the following promptline:

>C(opy: B(uffer F(rom file <esc>

To copy the text in the copy buffer, type 'B'. The Editor immediately copies the contents of the copy buffer into the file, starting from the location of the cursor when 'C' was typed. Use of the C(opy command does not change the contents of the copy buffer.

After the C(opy, the cursor is placed immediately after the text which was copied.

The copy buffer is affected by the following commands:

1) D(elete: On accepting a deletion, the buffer is loaded with the deletion; on escaping from a deletion, the buffer is loaded with what would have been deleted.

2) I(nsert: On accepting an insertion, the buffer is loaded with the insertion; on escaping from an insertion, the copy buffer is empty.

3) Z(ap: If the Z(ap command is used, the buffer is loaded with the deletion.

The copy buffer is of limited size. Whenever the deletion is greater than the buffer available, the Editor will issue the following warning:

There is no room to copy the deletion. Do you wish to delete anyway? (y/n)

A 'Y' or 'y' is a yes answer; any other character escapes D(elete.

To copy text from another file, type 'F' and another prompt appears:

>C(opy: From what file[marker,marker]?

Any file may now be specified; .TEXT is assumed. The markers (in brackets -- '[]') are optional, and used for copying only part of a file.

To copy part of a file, markers must be preset in that file to bracket the desired text. Two markers can be used, or the file's beginning or end may be part of the bracket. If [,marker] or [marker,] is used in C(opy, the file will be copied from the start of the file to the marker, or from the marker to the end of the file. Use of C(opy does not change the contents of the file being copied from.

IV.3.3 D(elete

On the promptline: D(el.

Repeat factors not allowed.

After entering D(elete, the following promptline appears:

>Delete: <> <moving commands> {<etx> to delete, <esc> to abort}

The cursor must be positioned at the first character to be deleted. On typing 'D' and entering D(elete, the Editor remembers where the cursor is. That position is called the anchor. As the cursor is moved from the anchor using the normal moving commands, text in its path disappears. Within D(elete, all cursor-moving commands are valid, including repeat factors and changes of direction.

Backing up over portions of the deletion restores those characters to the textfile.

To accept the deletion, type <etx>; to escape, type <esc>.

EXAMPLE:

Figure IV.8

```
PROGRAM STRING2;
BEGIN
  WRITE('TOO WISE ');
  WRITELN('TO BE.')
END.
```

Figure IV.9

```
PROGRAM STRING2:
BEGIN
END.
```

In Figure IV.8:
1) Move the cursor to the 'E' in END.
2) Type '<' (This changes the direction to backward)
3) Type 'D' to enter D(elete.
4) Type <return><return>. After the first return the cursor moves to before the 'W' in WRITELN, and 'WRITELN('TO BE.');' disappears. After the second return the cursor is before the 'W' in WRITE, and that line has disappeared.
5) Now press <etx>. The program after deletion appears as shown in Figure IV.9.

The two deleted lines have been stored in the copy buffer and the cursor has returned to the anchor position. Now C(opy may be used to copy the two deleted lines at any place to which the cursor is moved.

IV.3.4 F(ind

On the promptline: F(ind.

Repeat factors are allowed.

On entering Find, one of the promptlines in Figure IV.10 appears:

Figure IV.10

```
==============================
>Find[n]: L(it <target>  =>    { Which line appears depends
                                on the global mode (see S(et) }
>Find[n]: T(ok <target>  =>
==============================
```

(Where 'n' is the repeat factor given before typing 'F' for F(ind; this number is one if no repeat factor was given.)

F(ind finds the n-th occurrence of the <target> string, starting from the cursor's position and moving in the global direction (shown by the arrow at the beginning of the promptline). The cursor is positioned immediately after this occurrence.

If you desire to search in other than the global mode (either Token or Literal), type the appropriate character (either 'L' or 'T', respectively), before you enter the target string.

If the string is not present, the prompt:

ERROR: Pattern not in the file. Please press <spacebar> to continue.

... appears.

Example 1: In the STRING1 program (shown in Figure IV.11), with the cursor at the first 'P' in 'PROGRAM STRING1', type 'F'. When the prompt appears type "'WRITE'". The single quote marks must be typed. The promptline should now be:

>Find[1]: L(it <target> =>'WRITE'

Immediately after typing the last quote mark, the cursor jumps to the character following the 'E' in the first 'WRITE'.

Example 2: In the STRING1 program with the cursor at the 'E' of 'END.' type: '<3F'. This will find the third occurrence of the pattern in the reverse direction. When the promptline appears type '/WRITELN/'. The promptline should read:

<Find[3]: L(it <target> =>/WRITELN/

The cursor will move to immediately after the 'N' in WRITELN.

Figure IV.11

```
==========================================================================
PROGRAM STRING1;
BEGIN
  WRITE('TOO WISE ');          {cursor ends here in Example 1}
  WRITE('YOU ARE');            {cursor ends here in Example 3}
  WRITELN(',');                {cursor ends here in Example 2}
  WRITELN('TOO WISE ');
  WRITELN('YOU BE.')
END.                           {cursor starts here in Example 2}
==========================================================================
```

Example 3: On the first find we type 'F/WRITE/'. This locates the first 'WRITE'. Now typing 'FS' will make the promptline flash:

>Find[1]: L(it <target> =>S

... and the cursor will appear after the second WRITE.

IV.3.5 I(nsert

On the promptline: I(nsert.

Repeat factors not allowed.

On entering I(nsert, the following promptline appears:

>Insert: Text {<bs> a char,<del> a line} [<etx> accepts, <esc> escapes]

Characters are entered into the textfile as they are typed, starting from the position of the cursor. This includes the character <return>. Non-printing characters are echoed with the non-printing character symbol (usually a '?'; this can be changed by using SETUP). To make corrections while still in I(nsert, use <backspace> (<bs>) to remove one character at a time, or <rubout> (<del>) to remove an entire line. If you try to backspace past the beginning of the insertion, you will receive an error message.

The textfile that is actually created as you use I(nsert is to some extent dependent on the modes you have selected with the S(et E(nvironment commands. S(et E(nvironment is the means for selecting the Auto-indent and the Filling options.

IV.3.5.1 Using Auto-indent

If Auto-indent is True, a <return> causes the cursor to start the next line with an indentation equal to the indentation of the line above. If Auto-indent is False, a <return> returns the cursor to the first position of the next line. If Filling is True, the first position is the left margin (or the paragraph margin; see immediately below), otherwise it is the left-hand side of the screen.

IV.3.5.2 Using Filling

If Filling is True, the Editor forces all insertions to be between the right and left margins. It does this by automatically inserting <return>'s between "words" whenever the right margin would have been exceeded, and by indenting to the left margin whenever a new line is started. The Editor considers anything between two spaces, or between a space and a hyphen, to be a word.

A new paragraph is created when two <return>'s are typed in succession. In other words, a paragraph is a block of text delimited by blank lines (or command lines (see S(et), or the beginning or end of the textfile). The first line of a paragraph may be indented differently than the remaining text (see S(et E(nvironment).

If both Auto-indent and Filling are True, Auto-indent controls the Left-margin while Filling controls the Right-margin. The level of indentation may be changed by using the <space> and <backspace> keys immediately after a <return>. Important: This can only be done immediately after a <return>.

Example 1: With Auto-indent true, the following sequence creates the indentation shown in Figure IV.12.

'ONE'<return><space><space>'TWO'
<return>'THREE'<return><backspace>'FOUR'

Figure IV.12

```
ONE       original indentation
  TWO     indentation changed by <space> <space>
  THREE   <return> causes auto-indentation to level of line above
 FOUR     <backspace> changes indentation from level of line above
```

Example 2: With Filling True (and Auto-indent False) the following sequence creates the indentation shown in Figure IV.13:

'ONCE UPON A TIME THERE- WERE'.

(Very narrow margins have been used for simplicity.)

Figure IV.13

```
================================================================
ONCE UPON A       Auto-returned when next word would exceed margin
TIME THERE-       Auto-returned at hyphen
WERE
          ^
    Level of left margin
================================================================
```

The cursor may be forced to the left margin of the screen by typing <control-Q> (ASCII DC1).

Filling also causes the Editor to adjust the margins on the portion of the paragraph following the insertion. Any line beginning with the Command character (see S(et) is not affected by this adjustment, and such a line is considered to terminate a paragraph.

A filled paragraph may be re-adjusted by using the M(argin command. See Section IV.3.8. This may be very useful if the user wishes to change the margins of a document (which may be done with S(et E(nvironment).

The global direction does not affect I(nsert, but is indicated by the direction of the arrow on the promptline.

If an insertion is made and accepted, that insertion is available for use in C(opy. However, if <esc> is used, there is no string available for C(opy.

IV.3.6 J(ump

On the promptline: J(ump.

Repeat factors not allowed.

On entering J(ump, the following promptline appears:

>JUMP: B(eginning E(nd M(arker <esc>

Typing 'B' (or 'E') moves the cursor to the beginning (or the end) of the file. Typing 'M' causes the Editor to display the promptline:

Jump to what marker?

Markers are user-defined names for positions in the textfile. See the M(arkers option of the S(et command for more details.

IV.3.7 K(olumn

On the promptline: K(ol.

Repeat factors are allowed.

K(ol displays the following prompt:

>Kolumn: <vector keys> {<etx>, <esc> CURRENT line}

All of a line to the right of the cursor may be moved left or right by using <left-arrow> and <right-arrow>. Using <up-arrow> or <down-arrow> applies the same column adjustment to the line above (below).

Any characters at the cursor when K(olumn is used will be deleted by a <left-arrow>. The user should be careful not to delete things unintentionally.

IV.3.8 M(argin

On the promptline: M(argin.

Repeat factors not allowed.

F(illing must be set to "True," and A(uto indent must be set to "False" in the S(et E(nvironment template.

M(argin realigns the paragraph where the cursor is located to fit within the current margins. All of the lines within the paragraph are justified to the left margin, except the first line, which is justified to the paragraph margin. All these global margins may be set with the S(et E(nvironment command.

When you type 'M', the cursor may be located anywhere within the paragraph.

Example: The paragraph in Figure IV.14 has been M(argin'ed with the parameters on the left while the same paragraph in Figure IV.15 has been M(argin'ed with the parameters on the right.

```
Left-margin 0                    Left-margin 10
Right-margin 72                  Right-margin 70
Paragraph-margin 8               Paragraph-margin 0
```

Figure IV.14

```
        This quarter, the equipment is different, the course materials
are substantially different, and the course organization is different
from previous quarters.  You will be misled if you depend upon a friend
who took the course previously to orient you to the course.
```

Figure IV.15

```
This quarter, the equipment is different, the course materials are
          substantially different, and the course organization is
          different from previous quarters.  You will be misled if
          you depend upon a friend who took the course previously to
          orient you to the course.
```

A paragraph is any block of text delimited by blank lines or the beginning or end of the textfile. If the textfile or the paragraph is especially long, the screen may remain blank for several seconds while M(argin completes its work. When M(argin is done, the screen is redisplayed. M(argin never splits a word; it breaks lines at spaces or at hyphens.

IV.3.8.1 Command Characters

A line can be protected from being M(argin'ed by using the Command Character. The Command Character must be the first non-blank character in the line. M(argin (like Auto-fill) treats lines beginning with the Command Character as blank lines. The Command Character itself is any character so designated using the S(et E(nvironment command.

Note: If you use the M(argin command when in a line beginning with the Command character, M(argin will ignore the Command Character and M(argin the whole line, along with whatever is adjacent to it.

IV.3.9 P(age

On the promptline: P(age.

Repeat factors are allowed.

Moves the cursor one screenful in the global direction. The cursor remains on the same line on the screen, but is moved to the start of the line.

IV.3.10 Q(uit

On the promptline: Q(uit.

Repeat factors not allowed.

Q(uit displays the following prompt:

Figure IV.16

```
>Quit:
     U(pdate the workfile and leave
     E(xit without updating
     R(eturn to the editor without updating
     W(rite to a file name and return
```

One of the four options must be selected by typing U, E, R, or W. All other characters are ignored.

U(pdate:

Stores the file just modified as SYSTEM.WRK.TEXT, then leaves the Editor. SYSTEM.WRK.TEXT is the text portion of the workfile, and can be used as described in Section I.2.2.3, and chapters II and III.

E(xit:

This leaves the Editor immediately. Any modifications made since entering the Editor are not recorded in the permanent workfile. All editing during the session is irrecoverably lost, unless you have already used the W(rite option of Q(uit to save your work.

R(eturn:

Returns to the Editor without updating. The cursor is returned to the exact place in the file it occupied when "Q" was typed. This command is often used after unintentionally typing "Q". It is also useful when you wish to make a backup to your file in the middle of a session with the Editor.

W(rite:

This option puts up a further prompt:

Figure IV.17

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

The modified file may now be written to any filename. If it is written to the name of an existing file, the modified file will replace the old file. You may specify '$', which will update the file you have been editing. Q(uit can be aborted at this point by typing <return> instead of a filename; you will return to the Editor. If the file is written to disk, the Editor displays the following:

Figure IV.18

```
>Quit
Writing.....
Your file is 1978 bytes long.
Do you want to E(xit from or R(eturn to the Editor?
```

Typing 'E' exits from the Editor and returns to the System command level, while typing 'R' returns the cursor to the exact position in the file as when 'Q' was typed. Q(uit W(rite to '$' followed by R(eturn is a good way to back up your textfiles while you are working on them.

IV.3.11 R(eplace

On the promptline: R(plce.

Repeat factors are allowed.

On entering R(eplace one of the two promptlines in Figure IV.19 appears. In this example, a repeat factor of four is assumed:

Figure IV.19

```
=====================================
>Replace[4]: L(it V(fy <targ> <sub> =>   { Which one is used
                                           depends on the global
>Replace[4]: T(ok V(fy <targ> <sub> =>     mode (see S(et) }
=====================================
```

R(eplace finds the target string (<targ>) exactly as F(ind would, and replaces it with the substitution string (<sub>).

The verify option ('V(fy') permits examination of each <targ> string found in the text (up to the limit set by the repeat factor) so the user can decide if it is to be replaced. To use this option, type 'V' before typing the target string.

The following promptline appears whenever R(eplace has found the <targ> pattern in the file and verification has been requested:

>Replace: <esc> aborts, 'R' replaces, ' ' doesn't

Typing an 'R' at this point causes the replacement to take place, and the next target to be searched for. Typing a space causes the next occurrence of the target to be searched for. An <esc> at any point aborts the R(eplace.

With V(erify, this operation continues until the repeat factor is reached, or the target string can no longer be found.

With R(eplace in general, if the target string cannot be found, the prompt:

ERROR: Pattern not in the file. Please press <spacebar> to continue.

... appears.

R(eplace places the cursor after the last string which was replaced.

Example 1: Type 'RL/QX//YZ/'; the promptline appears as:

>Replace[1]: L(it V(fy <targ> <sub> =>L/QX//YZ/

This command will change: 'VAR SIZEQX:INTEGER;' to 'VAR SIZEYZ:INTEGER;'. Literal is necessary because the string QX is not a token, but part of the token SIZEQX.

Example 2: In Token mode, R(eplace ignores spaces between tokens when finding patterns to replace. For example, given the lines on the left-hand side of Figure IV.20, type "2RT/(',')/.LN." The promptline appears as:

>Replace: L(it V(fy <targ> <sub> =>/(',')/.LN.

Immediately after the last period was typed the two lines on the left of Figure IV.20 would change to those on the right-hand side.

Figure IV.20

```
=================================================================
    WRITE(',');                                  WRITELN;
    WRITE( ',');                                 WRITELN;
=================================================================
```

IV.3.12 S(et

On the promptline: S(et.

Repeat factors not allowed.

On entering S(et, the following promptline appears:

>Set: M(arker E(nvironment <esc>

IV.3.12.1 S(et M(arker

When editing, it is particularly convenient to be able to jump directly to certain places in a long file by using markers set in the desired places. Once a marker is set, it is possible to jump to it using the M(arker option in J(ump.

Move the cursor to the desired marker position, enter S(et, and type 'M' for M(arker. The following promptline appears:

Set what marker?

Markers may be given names of up to 8 characters followed by a <return>. Marker names are case-sensitive, so that lower and upper cases of the same letter are considered to be different characters. The marker will be entered at the position of the cursor in the text. If you use the name of a marker which already exists, it will be repositioned.

Only 20 markers are allowed in a file at any one time. If on typing "SM", the prompt:

Figure IV.21

```
==========================================================================
Marker ovflw.  Which one to replace? (Type in the letter or <sp>)

a) name1        b) name2        c) name3        d) name3
e)   .          f)   .          g)   .          h)   .
i)   .          j)   .          k)   .          l)   .
m)   .          n)   .          o)   .          p)   .
q) name17       r) name18       s) name19       t) name20
==========================================================================
```

... appears, it is necessary to eliminate one marker in order to replace it. Choose a letter (a through t), type that letter, and the specified marker will be eliminated, enabling you to set a new marker.

If a copy or deletion is made between the beginning of the file and the position of the marker, a J(ump to that marker may not subsequently return to the desired place, as the marker's absolute position has changed.

IV.3.12.2 S(et E(nvironment

The editing environment can be set to a mode which is most convenient for the editing being done -- whether on program text, document text, or data before processing. When in S(et type 'E' for E(nvironment; the screen display is replaced with the following prompt:

Figure IV.22

```
>Environment:  {options} <spacebar> to leave
     A(uto indent     True
     F(illing         False
     L(eft margin     9
     R(ight margin    70
     P(ara margin     9
     C(ommand ch      ^
     S(et tabstops
     T(oken def       true

     3152 bytes used, 29612 available.

Editing:  SCHEDULE.TEXT
Created November 11, 1982; last updated November 12, 1982 (revision 5).
Editor Version [IV.1 F6c].
```

(The parameters in this menu are samples, and will vary from file to file. The parameters shown for the letter options (e.g., C(ommand ch) are the default values.)

By typing the appropriate letter, any or all of the options may be changed.

IV.3.12.2.1 E(nvironment Options

A(uto indent:

Auto-indent affects only insertions. Refer to the section on I(nsert. Auto-indent is set to True (turned on) by typing 'AT' and to False (turned off) by typing 'AF'.

F(illing:

Filling affects I(nsert and M(argin. You should refer to those sections. Filling is set to True (turned on) by typing 'FT' and to False by typing 'FF'.

L(eft margin
R(ight margin
P(ara margin:

When Filling is True, the margins set in E(nvironment are the margins which affect I(nsert and M(argin. They also affect the Center and justifying commands in A(djust. To set a margin, type L, R, or P, followed by a positive integer and a <space>. The positive integer typed replaces the previous value. Margin values must be four digits or less.

C(ommand ch:

The Command character affects the M(argin command and the Filling option in I(nsert. Refer to those sections. Change the Command character by typing 'C' followed by any character. For example, typing 'C*' will change the Command character to '*'. This change will be reflected in the prompt. The Command Character was principally designed as a convenience for users of text formatting programs whose commands are indicated by a special character at the beginning of a line.

S(et tabstops:

Set tabs: <right, left vectors> C(ol# T(oggle tab <etx>

T----T----T----T----T----T----T----T----T----T----T----T----T----T----

Column#1

The cursor will start at position one in the line of Ts and dashes (-). The line 'Column#1' indicates the positon of the cursor. To set or remove a tab, move the

cursor to the desired location using the right or left vector keys, or press 'C' and enter the desired column number. Press 'T' to insert a tab or delete a tab. Pressing 'T' will change the indicator from a dash to T; pressing 'T' again in the same column will change the 'T' back to a dash. The system displays the current column number of the current cursor position and updates it each time a right/left vector key or 'C(olumn' command is pressed.

T(oken def:

This option affects F(ind and R(plc. Token is set to True by typing "TT" and to False by typing "TF". If Token is True, Token is the default and if Token is False, Literal is the default. See Section IV.2.6 for more information.

IV.3.13 V(erify

On the promptline: V(erify.

Repeat factors not allowed.

The current window is redisplayed, and the cursor is repositioned at the center of the text on the screen.

IV.3.14 eX(change

On the promptline: X(ch.

Repeat factors not allowed.

On entering eX(change the following promptline appears:

```
>eXchange: Text <vector keys> {<etx>,<esc>} CURRENT line}
```

Starting from the cursor position, eX(change replaces characters in the file with characters typed.

For example, in the file in Figure IV.23, with the cursor at the 'W' in WISE, typing 'XSM' replaces the 'W' with the 'S' and then the 'I' with the 'M', leaving the line as shown in Figure IV.24, with the cursor before the second 'S'.

Figure IV.23

```
====================
WRITE('TOO wISE ');
====================
```

Figure IV.24

```
====================
WRITE('TOO SMsE ');
====================
```

<etx> accepts the actions of eX(change, while <esc> leaves the command with no changes recorded in the last line altered.

eX(change ignores the global direction -- exchanges are always forward.

The arrow keys, <backspace>, <return>, and <tab> may be used to move the cursor about the screen. eX(changes move forward from wherever the cursor is moved to.

While in eX(change, the terminal's KEY TO INSERT CHARACTER inserts one space at the cursor's location, and KEY TO DELETE CHARACTER deletes a single character at the cursor's location. These keys are initially defined in the SYSTEM.MISCINFO file as the [SHIFT][I/R] and [SHIFT][-CHAR] keystrokes, respectively.

IV.3.15 Z(ap

On the promptline: Z(ap.

Repeat factors not allowed.

Deletes all text between the start of what was previously found, replaced, or inserted and the current position of the cursor. This command is designed to be used immediately after a F(ind, R(eplace or I(nsert. If more than 80 characters are being zapped, the Editor asks for verification.

The position of the cursor after the previous F(ind, R(eplace, or I(nsert is called the "equals mark". Typing '=' will place the cursor there.

Whatever was deleted by using the Z(ap command is available for use with C(opy, unless there is not enough room in the copy buffer. If this is the case, the Editor will ask if you want to Z(ap anyway.

After certain commands which might scramble the buffer, Z(ap is not allowed. These commands are: A(djust, D(elete, K(olumn, and M(argin.

V. YALOE -- YET ANOTHER LINE ORIENTED EDITOR

This text editor is intended for use on systems that do not have powerful screen terminals. It was designed to be very similar to the text editor which accompanies DEC's RT-11 system. Its name is pronounced: Yah-loo-ee.

To use YALOE, the user may eX(ecute YALOE.CODE. If extensive use of YALOE.CODE is planned, it is easier to use the Filer to C(hange SYSTEM.EDITOR to SCREEN.EDITOR (or some other name), and then C(hange YALOE.CODE to SYSTEM.EDITOR.

If there is a current workfile when YALOE is executed, it displays 'workfile STUFF read in'. If it does not find a workfile, it proclaims 'No work file to read'. This means that you entered YALOE with an empty workfile. From this point you may create a file in YALOE, and when you exit by typing 'QU', your workfile will no longer be empty.

YALOE operates in one of two modes: Command Mode or Text Mode. In command mode, all keyboard input is interpreted as commands instructing YALOE to perform some operation. When you first enter YALOE you will be in the Command Mode. The Text Mode is entered whenever the user types a command which must be followed by a text string. After the commands F(ind, G(et, I(nsert, M(acro define, R(ead file, W(rite to file, or eX(change all succeeding characters are considered part of the text string until an <esc> is typed. **Note:** when typed, <esc> echoes a '$'. The <esc> terminates the text string and causes YALOE to re-enter the Command Mode, at which point all characters are again considered commands.

Note: terminate command strings in YALOE with <esc><esc> to execute them. (This is unlike the rest of the System's 'immediate' commands.)

V.1 Special Key Commands

Various characters have special meanings, as described below. Some of these apply only in YALOE. Many have similar effects in the rest of the System; for these the ASCII code to which the System responds as indicated can be changed using the program SETUP, described in the Installation Guide.

<esc> — Echoes a '$'. A single <esc> terminates a text string. A double <esc> executes the command string.

[-LINE] <linedel> — Deletes current line. On hardcopy terminals, echoes '<ZAP'<return>. On others, it clears the current line on the screen. In both cases, the contents of that line are discarded by YALOE.

[BACK SPACE] <chardel> — Deletes character from the current line. On hardcopy terminals it echoes a '%', followed by the character deleted. Each succeeding CTRL H typed by the user deletes and echoes another character. A final '%' is printed when a key other than CTRL H is typed. This erasure is done right to left, up to the beginning of the command string. CTRL H may be used in both Command and Text mode.

CTRL X — Causes YALOE to ignore the entire command string currently being entered. YALOE responds with <return>* to indicate that the user may enter another command. For example:

```
*IDALE AND
KEITH<CTRL X>
*
```

A <linedel> deletes only KEITH; CTRL X erases the entire command.

All other control characters are ignored and discarded by YALOE.

V.2 Command Arguments

A commmand argument precedes a command letter and is used either to indicate the number of times the command should be performed, or to specify the particular portion of text to be affected by the command. With some commands, this specification is implicit and no argument is needed. Other commands, however, require an argument.

Command arguments are described as follows:

n — n stands for any integer. It may be preceded by a + or -. If no sign precedes n, it is assumed to be a positive number. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or -1 if only the - is present).

m — m is a number 0..9.

0 — '0' refers to the beginning of the current line.

/ — '/' means 32700. '-/' means -32700. It is useful as a large repeat factor.

= — '=' is used only with the J, D and C commands, and represents -n, where n is equal to the length of the last text argument used, for example: *GTHIS$=D$$ finds and removes THIS.

V.3 Command Strings

All EDIT command strings are terminated by two successive <esc>s. Spaces, carriage returns and tabs (CTRL I) within a command string are ignored unless they appear in a text string.

Several commands can be strung together and executed in sequence.

For example:

```
*B    GTHE INSERTED$   -3CING$      5K      GSTRING$$
```

The "B" sets the cursor position.

The "G" looks for the string "THE INSERTED" and places the cursor on the character which follows the "D".

The "-3CING" replaces the string "TED" with "ING".

The "5K" deletes text from the cursor to the 5th successive end-of-line.

The "GSTRING" finds the first occurance of "STRING" in the file and places the cursor just after the G.

As a rule, commands are separated from one another by a single <esc>. This separating <esc> is not needed, however, if the command requires no text. Commands are terminated by a single <esc>; a second <esc> signals the end of a command string, which will then be executed. When the execution of the command string is complete, YALOE prompts for the next command with '*'.

If an error is encountered at any point while executing the command, the command is terminated immediately. Any prior commands in the string will have already taken place.

V.4 The Text Buffer

The current version of the text is stored in the Text Buffer. This buffer's area is dynamically allocated; its size and the room left for expansion may be ascertained by using the ? command.

YALOE can only work on files that fit entirely within the Text Buffer.

V.5 The Cursor

The "cursor" is the position in your text where the next command will be executed. In other words it is the current "pointer" into the Text Buffer. Most edit commands refer to the cursor:

A,B,F,G,J: Moves it.
D,K: Remove text from where it is.
U,I,R: Add text to where it is.
C,X: Remove and then add text at it.
L,V: Print the text on the terminal from it.

V.6 Input/Output Commands

L(ist, V(erify, W(rite, R(ead, Q(uit

The L(ist command prints the specified number of lines on the console terminal without moving the cursor.

*-2L$$	Prints all characters starting at the second preceding line and ending at the cursor.
*4L$$	Prints all characters beginning at the cursor and terminating at the 4th <cr>.
*0L$$	Prints from the beginning of the current line up to the cursor.

The V(erify command prints the current text line on the terminal. The position of the cursor within the line has no effect and the cursor is not moved. Arguments are ignored. The V(erify command is equivalent to a 0LL (list) command.

The W(rite command is of the form

*W<filename>$

<filename> is any legal filename as decribed in Section 1.2, without the file type suffix. YALOE automatically appends a '.TEXT' suffix to the filename given, unless it ends with '.', ']', or '.TEXT'. If the filename ends in a '.', the dot will be stripped from the filename. Refer to Figure 4 (in Section III.5) for details on filename specifications.

The W(rite command writes the entire Text Buffer to a file with the given filename. It does not move the cursor or alter the contents of the Text Buffer.

If there is no room for the Text Buffer on the volume specified in the filename, the message:

OUTPUT ERROR. HELP!

... will be printed. It is still possible to write the Text Buffer by writing it to another volume.

The R(ead command is of the form

*R<file title>$

YALOE attempts to read the file title as given. In the event no file with that title is present, a '.TEXT' is appended and a new search is made.

The R(ead command inserts the specified file into the Text Buffer following the cursor. The cursor remains in the Text Buffer before the text inserted. If the file read in does not fit into the main memory buffer, the contents of the entire Text Buffer will be undefined, i.e. this is an unrecoverable error.

The Q(uit command has several forms

QU	Quit and update by writing out a new SYSTEM.WRK.TEXT
QE	Quit and escape session; do not alter SYSTEM.WRK.TEXT
QR	Don't quit; return to YALOE
Q	A prompt will be sent to the terminal giving all the above choices; enter option mnemonic (U, E, or R) only.

Executing the QU command is a special case of the write command, and the attempt to write out SYSTEM.WRK.TEXT may fail. In this case use the W command to write out your file and then QE to exit YALOE.

The QR command is used on the occasions when a Q is accidentally typed, and you wish to return to YALOE rather than leave it.

V.7 Cursor Relocation Commands

J(ump, A(dvance, B(eginning, G(et, F(ind

When using character and line oriented commands, a positive (n or +n) argument specifies the number of characters or lines in a forward direction, and a negative argument the number of characters or lines in a backward direction. The Editor recognizes a line of text as a unit when it detects a <return> in the text.

Carriage return characters are treated the same as any other character. For example, assume the cursor is positioned as indicated in the following text (^ represents the current position of the cursor, and does not appear in actual use. It is shown here for clarification):

THERE WAS A CROOKED MAN^<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>

The J(ump command moves the cursor over the specified number of characters in the Text Buffer. The edit command -4J moves the cursor back 4 characters.

THERE WAS A CROOKED^ MAN<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>

The command 10J moves the cursor forward 10 characters and places it between the 'H' and the 'U'.

THERE WAS A CROOKED MAN<CR>
AND H^UMPTY DUMPTY FELL ON HIM<CR>

The A(dvance command moves the cursor a specified number of lines. The cursor is moved to the beginning of the last line.

Hence, the command 0A moves the cursor to the beginning of the current line.

THERE WAS A CROOKED MAN<CR>
^AND HUMPTY DUMPTY FELL ON HIM<CR>

The command -1A (or -A) moves the cursor back one line.

^THERE WAS A CROOKED MAN<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>

The B(eginning command moves the cursor to the beginning of the Text Buffer. Use /J to move to the end of the buffer.

Search commands are used to locate specific characters or strings of characters within the Text Buffer.

The G(et and F(ind commands are synonymous. Starting at the position of the cursor, the current Text Buffer is searched for the nth occurrence of a specified text string. A successful search leaves the cursor immediately after the nth occurrence of the text string if n is positive, and immediately before the text string if n is negative. An unsuccessful search generates an error message and leaves the cursor at the end of the Text Buffer for n positive, and at the beginning for n negative.

*BGSTRING$=J$$ This command string will look for the string STRING starting at the beginning of the Text Buffer; and if found it will leave the cursor immediately before it.

V.8 Text Modification Commands

I(nsert, D(elete, K(ill, C(hange, eX(change

The I(nsert command causes the Editor to enter the TEXT mode. Characters are inserted immediately following the cursor until an <esc> is typed. The cursor is positioned immediately after the last character of the insert. Occasionally, with large insertions, the temporary insert buffer becomes full. Before this happens, a message is printed on the console: 'Please finish'. In response, type two successive <esc>s. To continue, type I to return to the Text mode.

Note: If you forget to type the I command, the text you enter will be treated as commands!

The D(elete command removes a specified number of characters from the Text Buffer, starting at the position of the cursor. Upon completion of the command, the cursor's position is at the first character following the deleted text.

*-2D$$ Deletes the two characters immediately preceding the cursor.

*B$FHOSE $=D$$ Deletes the first string 'HOSE ' in the Text Buffer, since =D used in combination with a search command will delete the indicated text string.

The K(ill command deletes n lines from the Text Buffer, starting at the position of the cursor. Upon completion of the command, the cursor's position is the beginning of the line following the deleted text.

*2K$$ Deletes characters starting at the current cursor position and ending at (and including) the second <CR>.

*/K$$ Deletes all lines in the Text Buffer after the cursor.

The C(hange command replaces n characters, starting at the cursor, with the specified text string. Upon completion of the command, the cursor immediately follows the changed text.

*0CAPPLES$$ Replaces the characters from the beginning of the line up to the cursor with 'APPLES', (equivalent to using 0X).

*BGHOSE$=CLIZARD$$ Searches for the first occurrence of 'HOSE' in the Text Buffer and replaces it with 'LIZARD'.

The eX(change command exchanges n lines, starting at the cursor, with the indicated text string. The cursor remains at the end of the changed text.

*-5XTEXT$$ Exchanges all characters beginning with the first character on the 5th line back and ending at the cursor with the string 'TEXT'.

*0XTEXT$$ Exchanges the current line from the beginning to the cursor with the string 'TEXT', (equivalent to using 0C).

*/XTEXT$$ Exchanges the lines from the cursor to the end of the Text Buffer with the text 'TEXT', (equivalent to using /C or /D1).

V.9 Other Commands

S(ave, U(nsave, M(acro, N (macro execution) and '?'

The S(ave command copies the specified number of lines into the Save Buffer, starting at the cursor. The cursor position does not change, and the contents of the Text Buffer are not altered. Each time a S(ave is executed, the previous contents of the Save Buffer, if any, are destroyed. If executing the S(ave command threatens to overflow the Save Buffer, the Editor generates a message to this effect, and does not perform the save.

The U(nsave command inserts the entire contents of the Save Buffer into the Text Buffer at the cursor. The cursor remains before the inserted text. If there is not enough room in the Text Buffer for the Save Buffer, the Editor generates a message to this effect, and does not execute the unsave.

The Save Buffer may be cleared with the command 0U.

The M(acro command is used to define macros. A maximum of ten macros, identified by a digit in 0..9 preceding 'M', are allowed. The default number is 1. The M(acro command is of the form:

mM%command string%

This says to store the command string into Macro Buffer number m, where m is the optional digit 0..9. The delimiter, '%' in this example, is always the first character following the M command and may be any character which does not appear in the macro command string itself. The second occurrence of the delimiter terminates the macro.

All characters except the delimiter are legal Macro command string characters, including single <esc>s. All commands are legal in a macro command string. Example of a macro definition:

*5M%GBEGIN$=CEND BEGIN$V$%$$

This defines macro number 5. When macro number 5 is executed, it will look for the string 'BEGIN', change it to 'END BEGIN', and then display the change.

If an error occurs when defining a macro, the message

'Error in macro definition'

... is printed, and the macro must be redefined.

The execute macro command, N, executes a specified macro command string. The form of the command is:

nNm$

Here n is simply any command argument as previously defined; m is the macro number (a digit 0..9) to be executed. If m is omitted, 1 is assumed. Because the digit m is technically a command text string, the N command must be terminated by an <esc>.

Attempts to execute undefined macros cause the error message 'Unhappy macnum'. Errors encountered during macro execution cause the message 'Error in macro'. Errors encountered in macro command syntax cause the message 'Error in macro definition'.

The ? command prints a list of all the commands and the sizes of the Text Buffer, Save Buffer, and available memory left for expansion. It also lists the numbers of the currently defined macros.

VI. THE ADAPTABLE ASSEMBLERS

VI.0 Introduction

VI.0.1 Assembly Language Definition

An assembly language consists of symbolic names which can represent machine instructions, memory addresses, or program data. The main advantage of assembly language programming over machine coding is that programs can be organized in a more readable and hence easier to understand fashion.

An assembly language program (called source code) is translated by an assembler into a sequence of machine instructions (called object code). Assemblers can create either relocatable or absolute object code. Relocatable code includes information that allows a loader to place it in any available area of memory, while absolute code must be loaded into a specific area of memory. Symbolic addresses in programs that are assembled to relocatable object code are called relocatable addresses.

VI.0.2 Assembly Language Applications

Users of the UCSD p-System are interested in developing assembly language programs for one of two purposes:

a) assembly language procedures running under the control of a host program in Pascal or FORTRAN.

b) stand-alone assembly language programs for use outside of the operating system's environment.

The UCSD Adaptable Assembler, in conjunction with the system linker and some support programs, has been designed to meet these needs. The assembler is adaptable in the sense that different versions built around one adaptable kernel exist for each processor supported by the UCSD p-System; new versions can be quickly generated for any new 8 or 16 bit processors that are introduced.

The Adaptable Assembler is a one pass assembler modeled after The Last Assembler (TLA) developed at the University of Waterloo. The basic concept behind both the TLA and the UCSD Adaptable Assembler is the use of a central machine independent core that is common to all versions of the assembler. This central core is augmented with machine specific modules to handle the architecture of each target machine.

This chapter is intended to be used in conjunction with the Assembler ROM and HP 82928A System Monitor Reference Manual. For information concerning differences from the processor's standard software syntax, see Section VI.8.

VI.1 General Programming Information

VI.1.1 Object Code Format

VI.1.1.1 Byte Organization

A byte consists of eight bits. The bits may represent eight binary values, or a single character of data. The bits may also represent a one byte machine instruction or a number which is interpreted either as a signed two's complement number in the range of -128 to 127 or an unsigned number in the range of 0 to 255.

VI.1.1.2 Word Organization

A word consists of sixteen bits, or two adjacent bytes in memory. A word may contain a one word machine instruction, any combination of byte quantities, or a number which may be interpreted either as a signed two's complement number in the range of -32,768 to 32,767 or an unsigned number in the range of 0 to 65,535.

VI.1.2 Source Code Format

VI.1.2.1 Character Set

The following characters are used to construct source code:

- upper and lower case alphabetic: A..Z, a..z
- numerals: 0..9
- special symbols: | @ # $ % ^ & * ( ) < > ~ [ ] . , / ; : " ' + - = ? _
- space (' ') character and tab character

VI.1.2.2 Identifiers

Identifiers consist of an alphabetic character followed by a series of alphanumeric characters and/or underscore characters. The underscore character is not significant. This definition of identifiers is equivalent to the Pascal definition.

Identifiers are used in:

- label and constant definitions.
- machine instructions, assembler directives, and macro identifiers.
- label and constant references.

Example: FormArray
FORM_ARRAY
formarray
... all denote the same item.

VI.1.2.2.1 Predefined Symbols and Identifiers

Predefined identifiers are reserved by the assembler as symbolic names for machine instructions and registers. They may not be used as names for labels, constants, or procedures. Also, the dollar sign ($) is predefined to specify the location counter. When used in an expression, the dollar sign represents the current value of the location counter in the program.

VI.1.2.3 Character Strings

A character string is written as a series of ASCII characters delimited by double quotes. A string may contain up to eighty characters, but cannot cross source lines. A double quote may be embedded in a character string by entering it twice, e.g., "This contains ""embedded"" double quotes". The assembler directive .ASCII requires a character string for its operand. Strings also have limited uses in expressions.

VI.1.2.4 Constants

VI.1.2.4.1 Binary Integer Constants

A binary integer constant is a series of bits or binary digits (0..1) followed by the letter 'T'. The range of values is 0 to 11111111, or 0 to 1111, if a byte constant.

Examples: 0T 01000100T 11101T

VI.1.2.4.2 Decimal Integer Constants

A decimal integer word constant is written as a series of numerals (0..9) followed by a period. Its range of values is -32768 to 32767 as a signed two's complement number. As a byte constant, its range of values is -128 to 127 as a signed two's complement number or 0 to 255 as an unsigned number.

Examples: 001. 256. -4096.

VI.1.2.4.3 Hexadecimal Integer Constants

A hexadecimal integer word constant is written as a series of up to four significant hexadecimal numerals (0..9, A..F) followed by the letter 'H'. The leading numeral of a hex constant must be a numeric character. The range of values is 0 to FFFF. These are examples of valid hex constants:

```
0AH
100H
0FFFEH              ; leading zero is required here
```

Byte constants possess similar syntax, but can have at most two significant hex numerals, with a range of 0 to FF.

VI.1.2.4.4 Octal Integer Constants

An octal integer word constant is written as a series of up to six significant octal numerals (0..7) followed by the letter 'Q'. Its range of values is 0 to 177777. Byte constants can have at most three significant octal numerals, with a range of 0 to 477.

Examples: 17Q 457Q 177776Q

VI.1.2.4.5 Default Radix Integer Constants

The radix of an integer constant lacking a trailing radix character is octal for the HP-86/87 assembler.

VI.1.2.4.6 Character Constants

Character constants are special cases of character strings and may be used in expressions. The maximum length is two characters for a word constant, and one character for a byte constant.

Examples: "A" "BC" "YA"

VI.1.2.4.7 Assembly Time Constants

An assembly time constant is written as an identifier that has been assigned a constant value by the .EQU directive (Section VI.2.2). Its value is completely

determined at assembly time from the expression following the directive. Assembly time constants must be defined before they may be referenced.

VI.1.2.5 Expressions

Expressions are used as symbolic operands for machine instructions and assembler directives. An expression can be:

- a label, which might refer to a defined address or an address further down in the source code (implying that the label is presently undefined), an externally referenced address, or an absolute address.
- a constant.
- a series of labels or constants separated by arithmetic or logical operators.
- the null expression, which evaluates to a constant of value 0.

VI.1.2.5.1 Relocatable and Absolute Expressions

An expression containing more than one label is valid only if the number of relocatable labels added to the expression exceeds the number of relocatable labels subtracted from the expression by zero or one. The expression result is absolute if the difference is zero, and relocatable if the difference is one. Subexpressions that evaluate to relocatable quantities may not be used as arguments to a multiplication, division, or logical operation. Unary operators may not be applied to relocatable quantities.

In relocatable programs, absolute expressions may not be used as operands of instructions which require location-counter-relative address modes.

VI.1.2.5.2 Linking and One Pass Restrictions

An expression may contain no more than one externally defined label, and its value must be added to the expression. An expression containing an external reference may not contain a forward referenced label, and the relocation sum of any other relocatable labels in the expression must be equal to zero.

An expression may contain no more than one forward referenced identifier. A forward referenced identifier is assumed to be a relocatable label defined further down in the source code; any other identifiers must be defined before they are used in an expression. An expression containing a forward referenced label may not contain an externally defined label.

VI.1.2.5.3 Arithmetic and Logical Operators

The following operators are available for use in expressions:

unary operations:
- '+' plus
- '-' minus (two's complement negation)
- '~' logical not (one's complement negation)

binary operations:
- '+' plus
- '-' minus
- '~' exclusive or
- '*' multiplication
- '/' signed integer division (DIV)
- '//' unsigned integer division (DIV)
- '%' unsigned remainder division (MOD)
- '|' bitwise OR
- '&' bitwise AND

The following operators are available for use only with conditional assembly directives:

- '=' equal
- '<>' not equal

The following symbols may be used as alternatives to the single character definitions presented above. Occurrences of these alternative definitions require at least single blank characters as delimiters.

```
.OR     =   '|'
.AND    =   '&'
```

```
.NOT    =    '~'
.XOR    =    '^'
.MOD    =    '%'
```

The assembler performs left to right evaluation of expressions; there is no operator precedence. All operations are performed on word quantities. Usage of unary operators is limited to constants and absolute addresses. Angle brackets must enclose subexpressions which contain embedded unary operators.

VI.1.2.5.4 Subexpression Grouping

Angle brackets ('<' and '>') may be used in expressions to override the left to right evaluation of operands. Subexpressions enclosed in angle brackets are completely evaluated before inclusion in the rest of the expression.

VI.1.2.5.5 Examples

The following are examples of valid expressions. The default radix is decimal.

```
MARK+4              ; The sum of the value of identifier MARK plus 4

BILL-2              ; The result of subtracting 2 from the value
                    ; of identifier BILL.

2-BARRY             ; The result of subtracting the value of
                    ; identifier BARRY from 2.  BARRY must be
                    ; absolute.

3*2+MACRO           ; The sum of the value of identifier MACRO plus
                    ; the product of 3 times 2.

DAVID+3*2           ; 2 times the sum of the identifier DAVID
                    ; and 3.  DAVID must be absolute.

650/2-RICH          ; The result of dividing 650 by 2 and sub-
                    ; tracting the value of identifier RICH from
                    ; the quotient.  RICH must be absolute.

                    ; Null expression: result is constant 0

-4*12+<6/2>         ; evaluates to -45 (decimal)

85+2+<-5>           ; evaluates to 82 (decimal)

0|1&<~0>            ; evaluates to 1

0 .OR 1 .AND <.NOT 0>  ; is the same expression (result is 1)
```

VI.1.3 Source Statement Format

An assembly language source program consists of source statements which may contain machine instructions, assembler directives, comments, or nothing (a blank line). Each source statement is defined as one line of a textfile. Assembly language identifiers are restricted to upper case alphabetic characters, but lower case characters may be used in the comment field.

VI.1.3.1 Label Field

The assembler supports the use of both standard labels and local (i.e., reuseable) labels. The label field begins in the leftmost character position of each source line. Macro identifiers and machine instructions must not appear in the start of the label field, but assembler directives and comments may appear there.

VI.1.3.1.1 Standard Label Usage

A standard label is an identifier that appears in the label field of a source statement. It may be terminated by an optional colon character, which is not used when referencing the label. As in Pascal, only the first eight characters of the label are important; the rest are ignored by the assembler. As in Pascal, the underscore character is not significant.

Example:

```
BIOS
L3456:          ; referenced as 'L3456'
The_Kind
LONG_label      ; last character is ignored
```

A standard label is a symbolic name for a unique address or constant; it may be declared only once in a source program. A label is optional for machine instructions and for many of the assembler directives. A source statement consisting of only a label is a valid statement; it has the effect of assigning the current value of the location counter to the label. This is equivalent to placing the label in the label field of the next source statement that generates object code. Labels defined in the label field of the .EQU directive (Section VI.2.2) are assigned the value of the expression in the operand field.

VI.1.3.1.2 Local Label Usage

Local labels allow source statements to be labeled for reference by other instructions without taking up storage space in the symbol table. They can contribute to the cleanliness of source program design by allowing the creation of nonmnemonic labels for use by iterative and decision constructs, thus reserving the use of mnemonic label names for demarking conceptually more important sections of code.

Local labels must have "$" in the first character position; the remaining characters must be digits. As in regular labels, only the first eight digits are significant. The scope of a local label is limited to the lines of source statements between the declaration of consecutive standard labels; thus, the jump to label $4 in the following example is illegal:

```
LABEL1

          CLB    R36
$3        STB    R36, R37
          JPS    $3            ; legal use of local label
          NOP
          JPS    $4            ; illegal use
LABEL2
          CLB    R46
$4        STB    R46,R56
```

Up to 21 local labels may be defined between 2 occurrences of a standard label. On encountering a standard label, the assembler purges all existing local label definitions; hence, all local label names may be redefined after that point. Local labels may not be used in the label field of the .EQU directive (Section VI.2.2).

VI.1.3.2 Opcode Field

The opcode field begins with the first non-blank character following the label field, or with the first nonblank character following the leftmost character position when the label is omitted. It is terminated by one or more blanks. The opcode field contains an identifier which can be of the following types:

- machine instruction
- assembler directive
- macro call

VI.1.3.3 Operand Field

The operand field begins with the first nonblank character following the opcode field, and is terminated by zero or more blanks. It can contain zero or more expressions, depending on the requirements of the preceding opcode.

VI.1.3.4 Comment Field

The comment field can be preceded by zero or more blanks, begins with a semicolon (';'), and extends to the end of the current source line. It may contain any printable ASCII characters. The comment field is listed on assembled listings, and has no other effect on the assembly process.

VI.1.4 Source File Format

Assembly source files are generated using the system editor and saved as files of type TEXT. A source file is constructed from the following entities:

- assembly routines (procedures and functions).
- global declarations.

VI.1.4.1 Assembly Routines

A source file may contain more than one assembly routine; in this case, a routine ends upon the occurrence in the source code of another program delimiting directive (i.e., the start of the following routine). Each routine in a source file is a separate entity; it contains its own relocation information and may be individually referenced by a Pascal host program during linking.

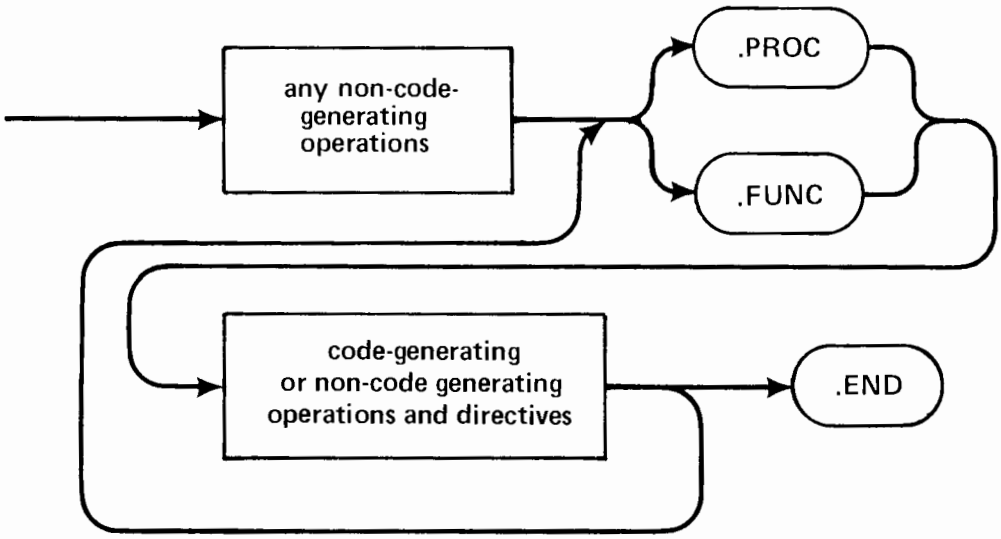
Assembly routines must begin with a .PROC, .FUNC, .RELPROC, or .RELFUNC directive. The last routine in the source file must be terminated by the .END directive. Section VI.5 gives a detailed description of these directives.

At the end of each routine, the assembler's symbol table is cleared of all but predefined and globally declared symbols, and the location counter (LC) is reset to zero.

VI.1.4.2 Global Declarations

An assembly routine may not directly access objects declared in another assembly routine, even if the routines are assembled in the same source file; however, occasions arise when it is desirable for a set of routines to share a commmon group of declarations. Therefore, the assembler allows global data declarations.

Any objects declared before the first occurrence of a .PROC or .FUNC directive in a source file may be referenced by all subsequent assembly routines. No code may be generated before the first procedure delimiting directive; hence, the 'global' objects are limited to the non-code-generating directives (.EQU, .REF, .DEF, .MACRO, .LIST, etc.).



VI.1.4.3 Absolute Sections

Assembly language programmers often find it necessary to access absolute addresses in memory, regardless of where an assembly routine is loaded in memory. For instance, a program may need to access ROM routines. Absolute sections allow the user to define labels and data space using the standard syntax and directives, but with the extra ability to specify absolute (nonrelocatable) label addresses starting at any location in memory.

Absolute sections are initiated by the directive .ASECT (for absolute section) and terminated by the directive .PSECT (for program section, which is the default setting during assembly). When the .ASECT directive is encountered, the absolute section location counter (ALC) becomes the current location counter. The .ORG directive can be used to set the ALC to any desired value. Label definitions are nonrelocatable and are assigned the current value of the ALC. The data directives .WORD, .BLOCK, and .BYTE cause the ALC (instead of the regular LC) to be incremented.

Data directives in an absolute section cannot place initial values in the location.

specified as they can when used in the program section; thus, the absolute section serves as a tool for constructing a template of label - memory address assignments.

The equate directive (.EQU) may be used in an absolute section, but the labels are restricted to being equated only to absolute expressions. The only other directives allowed to occur within an absolute section are .LIST, .NOLIST, .END, and the conditional assembly directives.

Absolute sections may appear as global objects.

The following is a simple example of an absolute section:

```
            .ASECT              ; start absolute section

            .ORG    0DF00H      ; set ALC to DF00 hex

                                ; note - no data values assigned
                                ; label assignments below

DSKOUT      .BYTE               ; DSKOUT = DF00
DSKSTAT     .BYTE               ; DSKSTAT = DF01
CONS        .WORD               ; CONS = DF02
BLAGUE      .BLOCK  4           ; BLAGUE = DF04 (4 bytes)
REMOUT      .WORD               ; REMOUT = DF08
OFFSET      .EQU    REMOUT+2    ; OFFSET = DF0A

            .PSECT
```

VI.2 Assembler Directives

Assembler directives (sometimes referred to as pseudo-ops) enable the programmer to supply data to be included in the program and exercise control over the assembly process. The following directives are common to all assembler versions. Assembler directives appear in the source code as predefined identifiers preceded by a period (.).

The following metasymbols are used below in the syntax definitions for assembler directives:

- special characters and items in capital letters must be entered as shown.
- items within angle brackets (<>) are defined by the user.
- items within square brackets ([]) are optional.
- the word 'or' indicates a choice between two items.
- items in lower case letters are generic names for classes of items.

The following terms are names for classes of items:

b =
the occurrence of one or more blanks.

integer =
any legal integer constant as defined in Section VI.1.2.4.

label =
any legal label as defined in Section VI.1.3.1.

expression =
any legal expression as defined in Section VI.1.2.5.

value =
any label, constant, or expression. Its default value is 0.

valuelist =
a list of zero or more values delimited by commas.

identifier =
a legal identifier as defined in Section VI.1.2.2.

idlist =
a list of one or more identifiers delimited by commas.

id:integer list =
a list of one or more identifier-integer pairs separated by a colon and delimited by a comma. The colon:integer part is optional; its default value is 1.

comment =
any legal comment as defined in Section VI.1.3.4.

character string =
any legal character string as defined in Section VI.1.2.3.

file identifier =
any legal name for a Pascal text file.

Example:

[<label>] [b] .ASCII b <character string> [<comment>]

... indicates that a label may be included in the label field (but is not necessary), and that a character string must be included as an operand.

Small examples are included after each definition to supply the user with a reference to the specific syntax of the directive.

VI.2.1 Procedure-Delimiting Directives

Every source program (including those intended for use as stand-alone code files) must contain at least one set of procedure-delimiting directives. The most frequent use of the assembler is in assembling small routines intended to be linked with a host compilation unit. The directives .PROC and .FUNC identify and delimit assembly language procedures. .RELPROC and .RELFUNC identify and delimit dynamically relocatable procedures. Dynamically relocatable procedures may reside in the code pool, and are subject to more of the System's memory management strategies. Section VI.5 has a more detailed description of the use of these directives.

.PROC Identifies the beginning of an assembly language procedure. The procedure is terminated by the occurrence of another delimiting directive in the source file.

FORM: [b] .PROC b <identifier> [,<integer>] [<comment>]

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of words of parameters passed to this routine. The default is 0.

EXAMPLE: .PROC DLDRIVE,2

.FUNC Identifies the beginning of an assembly language function, which is expected (by the host compilation unit) to return a function result on top of the stack; otherwise, equivalent to the .PROC directive.

FORM: [b] .FUNC <identifier>[,<integer>] [<comment>]

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of words of parameters passed to this routine. The default is 0.

EXAMPLE: .FUNC RANDOM

.RELPROC Identifies the beginning of a dynamically relocatable assembly language procedure. Such assembly procedures must be position-independent (see Section VI.5). The procedure is terminated by the occurrence of another delimiting directive in the source file.

FORM: [b] .RELPROC b <identifier> [,<integer>]
[<comment>]

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of words of parameters passed to this routine. The default is 0.

EXAMPLE: .RELPROC POOF,3

.RELFUNC Identifies the beginning of a dynamically relocatable assembly language function which is expected (by the host compilation unit) to return a function result on the stack; otherwise, equivalent to the .RELPROC directive.

FORM: [b] .RELFUNC <identifier>[,<integer>]
[<comment>]

<identifier> is the name associated with the assembly function.

<integer> indicates the number of words of parameters passed to this routine. the default is 0.

EXAMPLE: .RELFUNC POOOF

.END Marks the end of an assembly source file.

FORM: [<label>] [b] .END

VI.2.2 Data and Constant Definition Directives

.ASCII Converts character strings to a series of ASCII byte constants in memory. The bytes are allocated in the order that they appear in the string. An identifier in the label field is assigned the location of the first character allocated in memory. If an odd number of data bytes are generated on a word addressed assembler version, an extra zero byte will be emitted to word align the LC.

FORM: [<label>] [b] .ASCII b <character string> [<comment>]

<character string> is any string of printable ASCII characters delimited by double quotes.

EXAMPLE: .ASCII "HELLO"

.BYTE Allocates and initializes values in one or more bytes of memory. Values must be absolute byte quantities. The default value is zero. An identifier in the label field is assigned the location of the first byte allocated in memory.

FORM: [<label>] [b] .BYTE b [valuelist] [<comment>]

```
EXAMPLE:  TEMP  .BYTE   4   ; code would be:  04 octal
          TEMP1 .BYTE       ; code would be:  00 octal
```

.BLOCK Allocates and initializes a block of consecutive bytes/words in memory (bytes for byte addressed processors, words for word addressed processors). A byte value must be an absolute quantity. The default value is zero. An identifier in the label field is assigned the location of the first byte/word allocated.

FORM: [<label>] [b] .BLOCK b <length>[,<value>]
[<comment>]

<length> is the the number of bytes to allocate with the initial value <value>.

EXAMPLE: TEMP .BLOCK 4,6H

the output code would be:

```
06 06 06 06        ;four bytes with value 06 hex
```

.WORD Allocates and initializes values in one or more consecutive words of memory. Values may be relocatable quantities. The default value is zero. An identifier in the label field is assigned the location of the first word allocated.

FORM: [<label>] [b] .WORD b <valuelist> [<comment>]

EXAMPLE: TEMP .WORD 0,2,,4

the output code would be:

```
0000
0002
0000          ; this is a default value.
0004
```

```
L1   .WORD  L2
```

the output code would be a word containing the address of the label L2.

.EQU Equates a value to a label. Labels may be equated to an expression containing relocatable labels, externally referenced labels, and/or absolute constants. The general rule is that labels equated to values must be defined before use. The exception to this rule is for labels equated to expressions containing another label. Local labels may not appear in the label field of an equate statement.

FORM: <label> [b] .EQU b <value> [<comment>]

EXAMPLE: BASE .EQU R6

VI.2.3 Location Counter Modification Directives

These directives affect the value of the location counter (LC or ALC) and the location in memory of the code being generated.

.ORG — If used at the beginning of an absolute assembly program, .ORG initializes the location counter to <value>. Used anywhere else, .ORG will generate zero bytes until the value of the location counter equals <value>.

FORM: [b] .ORG b <value> [<comment>]

EXAMPLE:
```
.ORG    1000H
```

.ALIGN — Outputs sufficient zero bytes/words to set the location counter to a value which is a multiple of the operand value (bytes are emitted for byte addressed processors, words are emitted for word addressed processors).

FORM: [b] .ALIGN b <value> [<comment>]

EXAMPLE:
```
.ALIGN  2
```

This aligns the LC to a word boundary. (HP-86/87 processor is byte-addressable).

VI.2.4 Listing Control Directives

These directives allow the user to exercise control over the format of the assembled listing file generated by the assembler. No code is generated by these directives, and their source lines do not appear on assembled listings. See Section VI.7 for a more detailed description of an assembled listing.

.TITLE Changes the title printed on the top of each page of the assembled listing. The title may be up to 80 characters long. The assembler will change the title to 'SYMBOLTABLE DUMP' when printing a symbol table; the title reverts back to its former value after the symbol table is printed. The default value for the title is ' '.

FORM: [b] .TITLE b <character string> [<comment>]

EXAMPLE: .TITLE "P-CODE INTERPRETER"

.ASCIILIST Print all bytes generated by the .ASCII directive in the code field of the list file, creating multiple lines in the list file if necessary. Assembly begins with an implicit .ASCIILIST directive.

FORM: [b] .ASCIILIST [<comment>]

EXAMPLE: .ASCIILIST

.NOASCIILIST Limit the printing of data generated by the .ASCII directive to as many bytes as will fit in the code field of one line in the list file.

FORM: [b] .NOASCIILIST [<comment>]

EXAMPLE: .NOASCIILIST

.CONDLIST List source code contained in the unassembled sections of conditional assembly directives.

FORM: [b] .CONDLIST [<comment>]

EXAMPLE: .CONDLIST

.NOCONDLIST Suppress the listing of source code contained in the unassembled sections of conditional assembly directives. Assembly begins with an implicit .NOCONDLIST directive.

FORM: [b] .NOCONDLIST [<comment>]

EXAMPLE: .NOCONDLIST

.NOSYMTABLE Suppress the printing of a symbol table after each assembly routine in an assembled listing.

FORM: [b] .NOSYMTABLE [<comment>]

EXAMPLE: .NOSYMTABLE

.PAGEHEIGHT Control the number of lines printed in an assembled listing between page breaks. Assembly begins with an implicit .PAGEHEIGHT 59 directive.

FORM: [b] .PAGEHEIGHT <integer> [<comment>]

EXAMPLE: .PAGEHEIGHT

.NARROWPAGE Limit the width of an assembled listing to 80 columns. The symbol table is printed in a narrow format, source lines are truncated to a maximum of 49 characters, and title lines on the page headers are truncated to a maximum of 40 characters.

FORM: [b] .NARROWPAGE [<comment>]

EXAMPLE: .NARROWPAGE

.PAGE Continue the assembled listing on the next page by sending an ASCII form feed character to the assembled listing.

FORM: [b] .PAGE

EXAMPLE: .PAGE

.LIST Enables output to the list file, if a listing is not already being generated. .LIST and .NOLIST can be used to examine certain sections of source and object code without creating an assembled listing of the entire program. Assembly begins with an implicit .LIST directive.

FORM: [b] .LIST

EXAMPLE: .LIST

.NOLIST Suppresses output to the list file, if it is not already off.

FORM: [b] .NOLIST

EXAMPLE: .NOLIST

.MACROLIST Specifies that all following macro definitions will have their macro bodies printed when they are invoked in the source program. Assembly begins with an implicit .MACROLIST directive. Section VI.4 has a detailed description of macro language.

FORM: [b] .MACROLIST

EXAMPLE: .MACROLIST

.NOMACROLIST Specifies that all following macro definitions will not have their macro bodies printed when they are invoked in the source program. Only the macro identifier and parameter list are included in the listing.

FORM: [b] .NOMACROLIST

EXAMPLE: .NOMACROLIST

.PATCHLIST List occurences of all back patches of forward referenced labels in the list file. Assembly begins with an implicit .PATCHLIST directive. Section VI.7 has a detailed description of back patches.

FORM: [b] .PATCHLIST

EXAMPLE: .PATCHLIST

.NOPATCHLIST Suppress the listing of back patches of forward references.

FORM: [b] .NOPATCHLIST

EXAMPLE: .NOPATCHLIST

VI.2.5 Program Linkage Directives

Linking directives enable communication between separately assembled and/or compiled programs. Section VI.5 has a detailed description of program linking.

.CONST Allows access to globally declared constants in the host compilation unit by the assembly procedure.

FORM: [b] .CONST <idlist> [<comment>]

Each <id> is the name of a global constant declared in the Pascal host.

EXAMPLE:

```
.CONST    LENGTH
```

.PUBLIC Allows variables declared in the global data segment of the host compilation unit to be referenced by an assembly language routine.

FORM: [b] .PUBLIC <idlist> [<comment>]

Each <id> is the name of a global variable declared in the Pascal host.

EXAMPLE:

```
.PUBLIC   I,J,LENGTH
```

.PRIVATE Allows an assembly language routine to store variables in the global data segment of the host compilation unit that are accessable only to the assembly language routine.

FORM: [b] .PRIVATE <id:integer list> [<comment>]

EXAMPLE:

```
.PRIVATE    PRINT,BARRAY:9
```

Each <id> is treated as a label defined in the source code. <integer> determines the number of words of space allocated for <id>.

.INTERP Allows an assembly language procedure to access code or data in the P-code interpreter. .INTERP is a predefined symbol for a processor dependent location in the resident interpreter code; offsets from this base location may be used to access any code in the interpreter. Correct usage of this feature requires a knowledge of the interpreter's jump vector for this location. Its domain is generally restricted to systems applications.

FORM: valid when used in <expression>

EXAMPLE:

```
EXECERR   .EQU   12          ; hypothetical routine offset

BOMBINT   .EQU   .INTERP+EXECERR
          JMP    BOMBINT
```

.REF Provides access to one or more labels defined in other assembly language routines.

FORM: [b] .REF <idlist> [<comment>]

EXAMPLE: .REF SCHLUMP

.DEF Makes one or more labels to be defined in the current routine available to other assembly language routines for reference.

FORM: [b] .DEF <idlist> [<comment>]

EXAMPLE: .DEF FOON,YEEN

VI.2.6 Conditional Assembly Directives

Section VI.3 has a detailed description of conditional assembly features.

.IF Marks the start of a conditional section of source statements.

FORM: [b] .IF <expression> [= or <> <expression>] [<comment>]

EXAMPLE: .IF Z80

.ENDC Marks the end of a conditional section of source statements.

FORM: [b] .ENDC [<comment>]

EXAMPLE: .ENDC

.ELSE Marks the start of an alternative section of source statements.

FORM: [b] .ELSE [<comment>]

EXAMPLE: .ELSE

VI.2.7 Macro Definition Directives

Section VI.4 has a detailed description of macro language.

.MACRO Indicates the start of a macro definition

FORM: [b] .MACRO <identifier> [<comment>]

<identifier> is used to invoke
the macro being defined.

EXAMPLE: .MACRO ADDWORDS

.ENDM Marks the end of a macro definition.

FORM: [b] .ENDM [<comment>]

EXAMPLE: .ENDM

VI.2.8 Miscellaneous Directives

.INCLUDE Causes the assembler to start assembling the file named as an argument of the directive; when the end of this file is reached, assembling resumes with the source code that follows the directive in the original file. This feature is useful for including a file of macro definitions or for splitting up a source program too large to be edited as a single text file. .INCLUDE may not be used in an included source file (i.e., nested use of the directive) and may not be used in a macro definition.

FORM: [b] .INCLUDE <file identifier> [b <comment>]

The comment field of the .INCLUDE directive must be separated from the file identifier by at least one blank character.

EXAMPLE: .INCLUDE MYDISK:MACROS

.ABSOLUTE Causes the following assembly routine to be assembled without relocation information. Labels become absolute addresses and label arithmetic is allowed in expressions. Usage is valid only before the occurrence of the first procedure delimiting directive. .ABSOLUTE must not be used when creating a Pascal external procedure. Section VI.5 has a detailed description of absolute code files.

FORM: [b] .ABSOLUTE [<comment>]

EXAMPLE: .ABSOLUTE

.ASECT Specifies the start of an absolute section. Section VI.1.4.3 has a detailed description of .ASECT.

FORM: [b] .ASECT [<comment>]

EXAMPLE: .ASECT

.PSECT Specifies the start of a program section, and is used to terminate an absolute section. Section VI.1.4.3 has a detailed description of .PSECT.

FORM: [b] .PSECT [<comment>]

EXAMPLE: .PSECT

.RADIX Sets the current default radix to the value of the operand. Allowable operands are: 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). Section VI.1.2.2.4 has a detailed description of radices. Initial default radix is 8 (octal).

FORM: [b] .RADIX <integer> [<comment>]

EXAMPLE:

```
.RADIX    10    ; decimal default radix
```

VI.3 Conditional Assembly

Conditional assembly directives are used to selectively exclude or include sections of source code at assembly time. Conditional sections are initiated with the .IF directive and terminated with the .ENDC directive, and may contain the .ELSE directive. Control over the inclusion of conditional sections is determined by the use of conditional expressions. Conditional sections may contain other conditional sections.

When the assembler encounters an .IF directive, it evaluates the associated expression to determine the condition value. If the condition value is false, the source statements following the directive are discarded until a matching .ENDC or .ELSE is reached. If the .ELSE directive is used in a conditional section, source code before the .ELSE is assembled if the condition is true, and source code after the .ELSE is assembled if the condition is false.

Overall syntax for a conditional section (using the metalanguage described in Section VI.2) is as follows:

```
.IF <conditional expression>
<source statements>
[.ELSE
<source statements>]
.ENDC
```

VI.3.1 Conditional Expressions

A conditional expression can take one of two forms: a single expression, or comparison of two character strings or expressions. The first form is considered false if it evaluates to zero; otherwise, it is considered true. The second form of conditional expression is comparison for equality or inequality (indicated by the symbols '=' and '<>', respectively).

VI.3.2 Example

```
.IF     LABEL1-LABEL2   ; arithmetic expression

    ; This code is assembled only if
    ; difference is zero

  .IF     %1 = "STUFF"  ; comparison expression

      ; This code is assembled only if outer condition
      ; is true and text of first macro parameter
      ; is equal to "STUFF".

  .ENDC                 ; terminate nested section

    ; This code is assembled if outer condition
    ; is true

.ELSE

    ; This code is assembled if first condition
    ; is false

.ENDC                   ; terminate outer section
```

VI.4 Macro Language

The assembler supports the use of a macro language in source programs. A macro language allows the programmer to associate a set of source statements with an identifying symbol; when the assembler encounters this symbol (known as a macro identifier) in the source code, it substitutes the corresponding set of source statements (known as the macro body) for the macro identifier, and assembles the macro body as if it had been included directly in the source program. A carefully designed set of macro definitions can be used in all source programs to simplify the development of assembly language routines.

Macro language is enhanced by including a mechanism for passing parameters (known as macro parameters) to the macro body while it is being expanding, allowing a single macro definition to be used for an entire class of subtasks.

Here is a simple example:

```
                        ; macro definition...
.MACRO  STRING          ; macro identifier is STRING
                        ; macro body
                        ; %1 and %2 are parameter declarations
.BYTE   %2              ; 2nd parameter is length byte
.ASCII  %1              ; 1st parameter is argument
.ENDM                   ; end macro definition
```

Further down in the source code...

```
STRING  "WRITE",5.          ; 1st macro call
                            ; parameters are '"WRITE"' and '5.'
STRING  "TYPE SPACE",10.    ; 2nd macro call
                            ; parameters are '"TYPE SPACE"'
                            ; and '10.'
```

This is what gets assembled...

```
.BYTE   5.              ; data string declarations
.ASCII  "WRITE"

.BYTE   10.
.ASCII  "TYPE SPACE"
```

VI.4.1 Macro Definitions

Macro definitions may occur anywhere in a source program and are delimited by the directives .MACRO and .ENDM. The macro identifier must be unique to the source program, except when the programmer is redefining a predefined machine instruction name as a macro identifier. A macro definition may not include another macro definition; however, it may include macro calls. Macro calls may be nested to a maximum depth of five levels. A macro definition must occur before any calls to that macro are assembled, but macro calls may be forward referenced within the bodies of other macro definitions.

VI.4.2 Macro Calls

Macro calls may occur anywhere in a source program that code may be generated. A macro call consists of a macro identifier followed by a list of parameters. The parameters are delimited by commas and terminated by a carriage return or semicolon. Upon encountering a macro call, source code is read from the text of the corresponding macro body. Macro parameters within the macro body are substituted with the text of the matching parameter listed after the macro identifier which initiated the call.

VI.4.3 Parameter Passing

Macro parameters are referenced in a macro body by using the symbol '%n' in an expression, where 'n' is a single nonzero decimal digit. Upon scanning this symbol, the assembler replaces it with the text of the n'th macro parameter. Please note that macro parameters are not expanded within the quotes of an ASCII data string.

Three cases are possible:

1) The parameter exists - make the substitution.

2) The n'th parameter doesn't exist in the parameter list being checked (less than n parameters were passed); a null string is substituted.

3) Another symbol of the form '%m' is encountered in the parameter list. If nested macro calls exist, the text of the m'th parameter at the next higher level of macro nesting is substituted; otherwise, the symbol itself is assembled.

Parameters are passed without leading and trailing blanks. All assembly symbols except macro calls may be passed as parameters.

The following is an example of parameter passing in macros:

```
   .MACRO DOS
   UNO    %2,UN
   ICB    %2
   .ENDM

   .MACRO UNO
   LDB    %1,%2
   LRB    %3
   .ENDM
```

In a program, the macro call...

```
   DOS    TROIS,DEUX
```

assembles as...

```
   LDB    DEUX,UN     ; UNO got UN directly, but had to
                      ; use DOS's 2nd param
   LRB                ; 3rd param doesn't exist
   ICB    DEUX        ; DOS used its own 2nd param
```

VI.4.4 Scope of Labels in Macros

A problem arises in the use of macro language when the definition of a macro body requires the use of branch instructions and thus the presence of labels. Declaring a regular label in a macro body is incorrect if the macro is called more than once, for the label would be substituted twice into the source program and flagged by the assembler as a previously defined label. Location-counter-relative addressing can be used, but is prone to errors in nontrivial applications. The solution is to generate labels that are local to the macro body; the assembler's local labels have this capability.

Local label names declared in a macro body are local to that macro; thus, a section of code that contains a local label $1 and a macro call whose body also has the local label $1 will assemble without errors (contrast this with what happens when two occurrences of $1 fall between two regular labels). This feature allows local labels to be used freely in macros without fear of conflicts with the rest of the program.

Note - the maximum of 21 local labels active at any instant still applies.

VI.4.4.1 Local Labels As Macro Parameters

The passing of local labels as parameters has a special property. Unlike other macro parameters, local labels are not passed as uninterpreted text. The scope of a local label passed in a macro call does not change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is passing an address to a macro which simulates a conditional branch instruction.

The following is an example of passing local labels as macro parameters:

```
    .MACRO EIN
    JZR     $1
    JNZ     %1
$1
    .ENDM
```

In a program, the code...

```
TWIE
    LDM     ICHI,NI
    EIN     $1
    RTN
$1
    JSB     SAN
```

assembles as...

```
TWIE
    LDM     ICHI,NI         ; looks confusing, but if listing
                            ; was off, result is what programmer
                            ; meant to occur.
    JZR     $1              ; this references macro local label.
    JNZ     $1              ; this references outside $1.
$1                          ; macro local label

    RTN
$1                          ; outside $1
    JSB     SAN
```

VI.5 Program Linking and Relocation

The Adaptable Assembler produces either absolute or relocatable object code that may be linked as required to create executable programs from separately assembled or compiled modules.

Program linking directives generate information required by the System Linker to link modules. Some of the advantages of linking are:

Long programs can be divided into separately assembled modules to avoid a long assembly, reduce the symbol table size, and encourage modular programming techniques.

Modules can be shared by other linked modules.

Utility modules can be added to the System Library for use as external procedures by a large number of programs.

Pascal programs can directly call assembly language procedures.

The assembler generates linker information in both relocatable and absolute code files. The System Linker accesses this information during the linking process and removes it from the linked code file.

Relocatable code includes information that allows a loader program to place it anywhere in memory, while absolute (also called core image) codefiles must be loaded into a specific area of memory to execute properly. Assembly procedures running in the Pascal system environment must always be relocatable; the loading and relocation process is performed by the interpreter at a load address determined by the state of the System.

Absolute code will not run under the p-System environment (under which high-level programs must run). Relocatable code can run under the p-System. Code segments which contain statically relocatable code remain in main memory throughout the lifetime of their host program (or unit), and are position-locked for that duration. Thus, relocatable code may maintain and reference its own internal data space (or spaces). In addition, statically relocatable code saves some space because its relocation information does not have to remain present throughout the life of the program.

The directives .PROC and .FUNC designate statically relocatable routines; .RELPROC and .RELFUNC designate dynamically relocatable routines. Code segments which contain dynamically relocatable code do not necessarily occupy the same location in memory throughout their host's lifetime, but are maintained in

the code pool along with other dynamic segments (mostly P-code), and may be swapped in and out of main memory while the host program (or unit) is running. Thus, dynamically relocatable code cannot maintain internal data spaces -- data which is meant to last across different calls of the assembly routine must be kept in host data segments using .PRIVATEs and .PUBLICs. (It is the programmer's responsibility to make sure that this is the case.)

EXAMPLES:

1. Data space is embedded in the code, but the code does not move:

```
.PROC   FOON
.WORD   SPACE
  ...
.END
```

2. The code moves, but data space is allocated in the host compilation unit's global data segment:

```
.RELPROC   FOON
.PRIVATE   SPACE
  ...
.END
```

3. **Wrong:** The code moves, and the data is embedded in the code, so the data is destroyed:

```
.RELPROC   FOON
.WORD      SPACE
  ...
.END
```

Code pool management is described in the Internal Architecture Guide.

VI.5.1 Program Linking Directives

This section describes overall usage of linking directives. All linking of assembly procedures involves word quantities; it is not possible to externally define and reference data bytes or assembly time constants. Arguments of these directives must match the corresponding name in the target module (a lower case Pascal identifier will match an upper case assembly name, and vice versa) and must not

have been used before their appearance in the directive; all following references to the arguments are treated by the assembler as special cases of labels. These external references are resolved by the linker and/or interpreter by adding the link time and run time offsets to the existing value of the word quantity in question; thus, any initial offsets generated by the inclusion of external references and constants in expressions are preserved.

VI.5.1.1 Pascal Host Communication Directives

The directives .CONST, .PUBLIC, and .PRIVATE allow the sharing of constants and data between an assembly procedure and its host compilation unit. See Section VI.5.2.2 for examples.

.CONST Allows an assembly procedure to access globally declared constants in the host compilation unit. All references to arguments of .CONST are patched by the Linker with a word containing the value of the host's compile time constant.

.PUBLIC Allows an assembly procedure to access globally declared variables in the host compilation unit. Note - this directive can be used to set up pointers to the start of multi-word variables in host programs; it is not limited to single word variables.

.PRIVATE Allows an assembly procedure to declare variables in the global data segment of the host compilation unit that are inaccessable to the host. The optional length attribute of the arguments allows multi-word data spaces to be allocated; the default data space is one word.

VI.5.1.2 External Reference Directives

The directives .REF and .DEF allow separately assembled modules to share data space and subroutines. See Section VI.5.2.2 for examples.

.DEF declares a label to be defined in the current program as accessable to other modules. One restriction is imposed on usage - it is invalid to .DEF a label that has been equated to a constant expression or an expression containing an external reference.

.REF declares a label existing and .DEF'ed in another module to be accessable to the current program.

VI.5.1.3 Program Identifier Directives

The directives .PROC, .FUNC, .RELPROC, .RELFUNC, and .END serve as delimiters for source programs. Every source program (relocatable or absolute) must contain at least one pair of delimiting directives (see Section VI.1.4.1).

The identifier argument of the .PROC or .RELPROC directive serves two functions: it is referenced by the Linker when linking an assembly procedure to its corresponding host, and it can be referenced as an externally declared label by other modules. Specifically, the declaration:

```
        .PROC FOON    ; procedure heading
```

... in a source program is functionally equivalent in the assembly environment to the following statements:

```
        .DEF FOON     ; FOON may be externally referenced
FOON                  ; declare FOON as a label
```

This feature allows an assembly module to call other (external and eventually linked in) assembly modules by name. The .FUNC and .RELFUNC directives are used when linking an assembly function directly to a Pascal host program; they are not intended for uses which involve linking with other assembly modules.

The optional integer argument after the procedure identifier is referenced by the Linker to determine if the number of words of parameters passed by the Pascal host's external procedure declaration matches the number specified by the assembly procedure declaration; it is not relevant when linking with other assembly modules.

VI.5.2 Linking Program Modules

For information on linking with the p-System's other high-level languages, please refer to the documentation on that particular language.

VI.5.2.1 Linking With a Pascal Host Program

External procedures and functions are assembly language routines declared in Pascal programs. In order to run Pascal programs with external declarations, it is necessary to compile the Pascal program, assemble the external procedure or function, and link the two codefiles. The linking process can be simplified by adding the assembled routine to the system library with the librarian program.

A host program declares a procedure to be external in a syntactically similar manner to a forward declaration. The procedure heading is given (with parameter list, if any), followed by the keyword 'EXTERNAL'. Calls to the external procedure use standard Pascal syntax, and the Compiler checks that calls to the external procedure agree in type and number of parameters with the external declaration. All parameters are pushed on the stack in the order of their appearance in the parameter list of the declaration; thus, the rightmost parameter in the declaration will be on the top of stack. Section VI.5.2.1.1 has a detailed description of parameter passing conventions.

It is the programmer's responsibility to assure that the assembly language routine maintains the integrity of the stack. This includes removing all parameters passed from the host, preserving any machine resources in use by the interpreter , and making a clean return to the Pascal run time environment using the return address originally passed to it. The price of nonconformance in these matters is a potentially fatal system crash, as assembly routines are outside the scope of the Pascal environment's run time error facilities.

An external function is similar to a procedure, but with some differences that affect the way in which parameters are passed to and from the Pascal runtime environment; first, the external function call will push one or two words on the stack (two for a function of type real, one for all other types) before any parameters have been pushed. The words are part of the P-machine's function calling mechanism, and are irrelevant to assembly language functions; the assembly routine must throw these away before returning the function's result. Second, the assembly routine must push the proper number of words (2 for type real, 1 otherwise) containing the function result onto the stack before passing control back to the host.

VI.5.2.1.1 Parameter Passing Conventions

The ability of external procedures to pass any variables as parameters gives the assembly programmer complete freedom to access the machine dependent representations of machine independent Pascal data structures; however, with this freedom comes the responsibility of respecting the integrity of the Pascal run time environment. This section attempts to enumerate the P-machine's parameter passing conventions for all data types in order that the programmer may gain a better understanding of the Pascal/assembly language interface; it does not actually describe data representations.

Parameters may be passed either by value or by name (also known as variable parameters). For purposes of assembly language manipulation, variable parameters are handled in a more straightforward fashion than value parameters.

The word 'tos' is used in the following sections as an abbreviation for 'top of stack'.

VI.5.2.1.1.1 Variable Parameters

Variable parameters are referenced through a one word pointer passed to the procedure. Thus, the procedure declaration:

```
procedure pass_by_name(var i,j : integer;
                                    var q : some_type);  external;
```

...would pass 3 one word pointers on the stack; tos would be a pointer to q, followed by pointers to j and i.

A Pascal external procedure declaration is allowed to contain variable parameters lacking the usual type declaration; this enables variables of different Pascal types to be passed through a single parameter to an assembly routine. Untyped parameters are not allowed in normal Pascal procedure declarations.

The procedure declaration:

```
procedure untyped_var(var i; var q : some_type);
      external;
```

...contains the untyped parameter i.

VI.5.2.1.1.2 Value Parameters

Value parameters are handled in a manner dependent upon their data type. The following types are passed by pushing copies of their current values directly on the stack: boolean, char, integer, real, subrange, scalar, pointer, set, and long integer. Other sections of the user manual describe the number of words per data type and the internal data format. For instance, the declaration:

```
procedure pass_by_value(i : integer; r : real); external;
```

...would pass 2 words on tos containing the value of the real variable r followed by one word containing the value of the integer variable i.

Variables of type record and array are passed by value in the same manner as variable parameters; pointers to the actual variable are pushed onto the stack. Variables of type PACKED ARRAY OF CHAR and STRING are passed by value with a segment pointer (described in the following paragraphs).

Pascal procedures protect the original variables by using the passed pointer to copy their values into a local data space for processing; assembly procedures should respect this convention and not alter the contents of the original variables.

VI.5.2.1.1.2.1 Accessing Byte Array Parameters with a Segment Pointer

A segment pointer consists of two words on the stack. The first word (tos) contains either NIL (0) or a pointer to a segment environment record.

If the first word is NIL, then the second word (at tos-1) points to the parameter.

If the first word is not NIL, then to find the parameter it is necessary to chain through some records. The first word is a pointer and the second word is an offset. The first word points to a segment environment record. The second word of this record contains a pointer to a pointer to the base of the segment where the parameter resides. The exact location of the parameter is given by the second word on the stack (tos-1), which is an offset into the code segment.

This address chain may be described as follows (offsets are word offsets):

(first word + 1) + <contents of second word>

A full description of these mechanisms may be found in the Internal Architecture Guide.

VI.5.2.2 Sample Assembly Language Program

The sample assembly language program and Pascal program listed below demonstrate how to pass parameters to and from external procedures and functions. You should be familiar with HP-86/87 assembly language as described in The Adaptable Assembler section of this manual and in the HP-86/87 Assembler ROM reference manual.

The following Pascal program declares three external assembly language routines.

```
program  DEMONSTRATE_ASSEMBLER;

var      Line : string;

procedure RUN_LITE (Turn_On : boolean); external;
```

```
procedure UPCASE (var S : string); external;

function  EXCLUSIVE_OR (X,Y : integer) : integer; external;

BEGIN
  RUN_LITE(true);          (*cause power light to blink*)


  writeln('255 exclusive or "ed with 256 is ',
               EXCLUSIVE_OR(255, 256));

  write('Enter a string of characters: ');

  readln(Line);

  UPCASE(Line);

  writeln('In upper case it is "' Line,'."');

  RUN_LITE(false);         (*stop blinking power light*)
END.
```

Enter the following assembly language program using the Editor, which will store it as a .TEXT file. Use the Assembler to assemble the .TEXT file into a .CODE file. Then use the Linker to link the compiled Pascal program and the assembled assembly language routines.

```
;External Procedures for Demonstration Program
;EMC PTR2 -- most significant byte of PTR2 always
;points to p-Machine data space.
PTR2      .EQU        177714          ;EMC PTR2
PTR2DEC   .EQU        177715          ;EMC PTR2 pre-decrement
PTR2INC   .EQU        177716          ;EMC PTR2 post-increment
PTR2DCIC  .EQU        177717          ;EMC PTR  pre-decrement,
                                      then post decrement

          .MACRO      POP             ;macro to pop data off
                                      ;evaluation stack
          POMD        %1,-R12
          .ENDM

          .MACRO      PUSH            ;macro to push data onto stack
          PUMD        %1,+R12
          .ENDM

          .RELPROC    RUN_LITE,1      ;has one word of parameters
                                      ;on stack
```

```
RULITE    .EQU        177704          ;light blinks if a 1 is stored
                                      ;at RULITE,
                                      ;doesn't blink if a 0 is there
          POP         R20
          ANM         R20,=1,0        ;mask all but least significant bit
          STBD        R20,RULITE      ;blink if R20=1, no-blink if R20=0
          RTN

          .RELEFUNC EXCLUSIVE_OR,2
          POP         R20             ;get first parameter
          POP         R22             ;get second parameter
          POP         R24             ;remove function result
          XRM         R20,R22
          PUSH        R20             ;push actual function result
          RTN

          .RELPROC UPCASE,1           ;one word of parameters
LENGTH    .EQU        R22
CHAR      .EQU        R23
          POP         R20             ;pointer to string
                                      ;in p-Machine data space
          STMD        R20,PTR2        ;set PTR2 to point to string
          LDBI        LENGTH,PTR2INC  ;get string length byte
          JZR         $30

$10       LDBI        CHAR,PTR2INC    ;get char in string
          CMB         CHAR,="a"       ;see if it's lowercase
          JNC         $20
          CMB         CHAR,="z"+1
          JCY         $20
          SBB         CHAR,=32.       ;convert to uppercase
          STBI        CHAR,PTR2DCIC   ;put char back into string

$20       DCB         LENGTH
          JNZ         $10

$30       RTN

          .END
```

VI.5.2.3 Stand-alone Applications

The Adaptable Assemblers were originally developed to allow the Pascal project to maintain all interpreters and I/O systems on the Pascal System in order to be completely self-supporting; thus, in their current configuration, the assemblers have the capability to produce absolute (core image) codefiles for use outside of the p-System's runtime environment.

The p-System does not include a linking loader or an assembly language debugger, as the P-machine architecture is not conducive to running programs (whether high or low level) that must reside in a dedicated area of memory. The user is responsible for loading and executing the object codefile; this can be done using the p-System, with the understanding that the existing runtime environment may be jeopardized in the process. Section VI.5.2.3.2 provides some ideas on how to create a Pascal loader program.

The utility COMPRESS is a much easier and more versatile way of doing this task. It allows for relocation and compaction of code. Refer to Section IX.1.

VI.5.2.3.1 Assembling

The .ABSOLUTE and .ORG directives are used to create an object codefile suitable for use as an absolute core image. .ABSOLUTE causes the creation of nonrelocatable object code, and .ORG may be used to initialize the location counter to any starting value. A source file headed by .ABSOLUTE should not have more than one assembly routine; sequential absolute routines do not produce continuous object code and cannot be successfully linked with one another to produce a core image.

The codefile format consists of a 1 block codefile header followed by the absolute code, and is terminated by one block of linker info; thus, stripping off the first and last block of the codefile will leave a core image file. The use of .ABSOLUTE should be limited to one routine; though linker information is generated, it is difficult to link absolute codefiles so as to produce a correct core image file.

VI.5.2.3.2 Loading and Executing Absolute Codefiles

The following section describes one method of loading and executing absolute codefiles using the UCSD p-System. The program outlined is not the only solution. It is also feasible to use the system intrinsics to read and/or move the codefile into the desired memory location, but this requires a knowledge of where the interpreter, operating system, and user program reside in order to prevent system crashes by accidentally overwriting them. The program outlined below

allows the most freedom in loading core images; the only constraint is that the assembly code itself is not overwritten while being moved to its final location. This possibility can be detected before loading proceeds.

It must be emphasized that in most cases loading object code into arbitrary memory locations while a Pascal system is resident will adversely affect the system; the absolute assembly language program is then on its own, and rebooting may be necessary to revive the Pascal system.

The loader program consists of:

1) A Pascal host program that calls two external procedures.

2) One or more linkable absolute codefiles to be loaded. (.RELPROCs are not allowed.)

3) A small assembly procedure MOVE_AND_GO that moves the above object codefiles from their system load address to their proper locations and transfers control to them.

4) A small assembly language procedure LOAD_ADDRESS that returns the system load addresses of the aforementioned assembly code to the host program.

The absolute codefiles are assembled to run at their desired locations, and MOVE_AND_GO contains the desired load addresses of each core image. Both LOAD_ADDRESS and MOVE_AND_GO have external references to the core images; these are used to calculate the system load address and code size of each image file. The whole collection is linked and executed, with the Pascal host performing the following actions:

Print the result of calling LOAD_ADDRESS to determine whether the area of memory in which the Pascal system loaded the assembly code overlays the known final load address of the core images. Issue a prompt to continue, so that the program can be aborted if a conflict does arise.

Call MOVE_AND_GO.

VI.6 Operation of the Assembler

The system assembler is invoked by typing 'A' at the command level of the operating system. This command will execute the file named SYSTEM.ASSMBLER (note the missing 'E' in the file name; this is required for conformance with the file system's restrictions on file name lengths); if this is not the name of the desired assembler version, be sure to save the existing file 'SYSTEM.ASSMBLER' under a different name before changing the desired assembler's name to 'SYSTEM.ASSMBLER'.

VI.6.1 Support Files

There are two associated support files: an opcodes file and an error file. These should always be stored along with the assembler code file.

In order for the assembler to run correctly, it is necessary that the proper opcodes file be present on some on-line disk; the assembler will search all units in increasing order of the unit number until it finds it. The opcode file must have the name HP86/87.OPCODES. The opcode file contains all predefined symbols (instructions and register names) and their corresponding values for the associated assembly language. If the proper opfile is not on-line, the assembler will write '<opfilename> not on any vol' and abort the assembly.

The assembler also has its own error file which contains a list of machine specific error messages. The error file must have the name HP86/87.ERRORS. The presence of an error file is not necessary for running the assembler, but can greatly aid the chore of squeezing the syntax errors out of a freshly written program.

Refer to appendix E for a listing of HP-86/87 assembler syntax errors.

VI.6.2 Setting Up Input and Output Files

When the assembler is first invoked from the prompt line, it will attempt to open the work file as its input file; if a work file exists, the first prompt will be the listing prompt described in Section VI.6.3 and the generated code file will be named ´SYSTEM.WRK.CODE´. If not, this prompt will appear:

```
Assemble what text?
```

Type in the file name of the input file followed by a carriage return. Typing only a carriage return will abort the assembly; otherwise, the next prompt will then appear:

```
To what codefile?
```

Type in the desired name of the output code file followed by a carriage return. Typing only a carriage return here will cause the assembler to name the output ´*SYSTEM.WRK.CODE´, but typing ´$´ will cause the code file to be created with the same filename prefix as the source file. The assembler will then display its standard listing prompt.

VI.6.3 Response to Listing Prompt

Before assembling begins, the following prompt will appear on the console:

```
HP 86/87 Assembler IV.1 [a.5]
Output file for assembled listing: (<CR> for none)
```

HP 86/87 is the processor number and IV.1 is the release level of the assembler. At this point, the user may respond with one of the following:

0) The escape key will abort the assembly and return the user to the operating system prompt.

1) 'CONSOLE:' or '#1:' will send an assembled listing of the source program to the screen during assembly.

2) 'PRINTER:' or '#6:' will send an assembled listing to the printer unit.

3) 'REMOUT:' or '#8:' will send an assembled listing to the REMOTE unit.

4) A carriage return will cause the assembler to suppress generation of an assembled listing and ignore all listing directives.

5) All other responses will cause the assembler to write the assembled listing to a text file of that name; any existing textfile of that name will be removed in the process. For instance, the following responses will cause a list file named 'LISTING.TEXT' to be created on disk unit 5:

```
#5:listing.text
#5:listing
```

In all cases, it is the responsibility of the user to ensure that the specified unit is on-line; the assembler will print an error message and abort if it is requested to open an off-line I/O unit.

VI.6.4 Output Modes

If the user sends an assembled listing to the console, logic dictates that this is what will be displayed on the screen during the assembly process; however, if the listing is sent to some other unit or if no listing is generated, the assembler writes a running account of the assembly process to the screen for the user's benefit. One dot is written to the screen for every line assembled; on every 50'th line, the number of lines currently assembled is written on the left hand side of the screen (delimited by angle brackets).

When an include file directive is processed by the assembler, the console displays the current source statement:

.INCLUDE <file name>

This allows the user to keep track of which include file is currently being assembled.

At the end of the assembly, the console displays the total number of lines assembled in the source program and the total number of errors flagged in the source program.

VI.6.5 Responses to Error Prompt

When the assembler uncovers an error, it will print the error number and the current source statement (if applicable to the error; this does not apply to undefined labels and system errors). It then attempts to retrieve and print an error message from the errors file. If the errors file cannot be opened (file is nonexistent or lack of memory), no message will appear. This is followed by the prompt:

E(dit, <space>, <esc>

Typing an 'E' will invoke the editor, a space will continue the assembly, and an escape character will abort the assembly. Some restrictions exist when either invoking the editor or attempting to continue:

1) In most cases, typing a space character restarts the assembly process with no problems; since assembly language source statements are independent of one another with respect to syntax, it is not a difficult task for the assembler to continue generating a code file. Thus, a code file will exist at the end of an assembly if the user types a space for every (nonfatal) error prompt that appears; of course, the code produced may not be a correct translation of the user's source program. Certain system errors are considered fatal by the assembler; these errors will abort the assembly regardless of the response given to the above prompt.

2) If an 'E' is typed, the system automatically invokes the editor, which opens the file containing the offending error and positions the cursor at the location where the error occurred. This feature will always work correctly when the source program is wholly contained in one file; however, when include files are used, the user should set up the input and output files manually (see Section VI.6.2) in order for the editor to position the cursor in the file that contains the error.

VI.6.5.1 Miscellany

At the end of an assembly, an error message for each undefined label is printed. In some cases, occurrences of undefined labels can be ignored by the user if the labels in question are semantically irrelevant to the desired execution of the code file; the resulting code file will be perfectly valid, but the references to the nonexistent labels will not be completely resolved.

In addition to generating a codefile, the assembler makes use of a scratch file, which is always removed from the disk upon normal termination of the assembly. Occasionally though, a system error may occur that will prevent the assembler from removing this file; if this happens, a new file may appear named 'LINKER.INFO'. It may be removed without anxiety, as it is entirely useless outside of the assembler's domain. This should be a rare (if not nonexistent) phenomenon.

VI.7 Assembler Output

The assembler can generate two varieties of output files. A codefile is always produced, but the user controls whether an assembled listing of the source file is produced.

An assembled listing displays each line of the source program, the machine code generated by that line, and the current value of the location counter. The listing may display the expanded form of all macro calls in the source program. Any errors that occur during the assembly process have messages printed in the listing file, usually immediately following the line of source code that caused the error. A symbol table is printed at the end of the listing; it serves as a directory for locating all labels declared in the source program.

An assembled listing of a source program printed on hard copy is one of the most effective debugging aids available for assembly language programs; it is equally useful for off-line, 'mental' debugging and in conjunction with system debuggers.

A description of the codefile format is beyond the scope of this document.

VI.7.1 Source Listing

A paginated assembled listing is produced when the user responds to the assembler's listing prompt with a listfile name. The default listing is 132 characters wide and 55 lines per page. Each line of a source program is included in the assembled listing, except for source lines that contain list directives. Source statements that contain the equate directive .EQU have the resulting value of the associated expression listed to the left of the source line.

Macro calls are always listed, including the list of macro parameters and the comment field, if any. The macro is expanded by listing the body (with all formal parameters replaced by their passed values) if the macro list option was enabled when the macro was defined. Macro expansion text is marked in the assembled listing by the character '#' just to the left of the source listing. Comment fields in the definition of the macro body are not listed in macro expansions.

Source lines with conditional assembly directives are listed; however, source statements in an unassembled part of a conditional section are not listed.

VI.7.2 Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console (see Section VI.6), except that the user prompt is not included.

VI.7.3 Code Listing

The code field lies to the left of the source program listing. It always contains the current value of the location counter, along with either code generated by the matching source statement or the value of an expression occuring in a statement that includes the equate directive .EQU; all are printed in octal. Separately emitted bytes and words of code on the same line are delimited by spaces.

VI.7.3.1 Forward References

When the assembler is forced to emit a byte or word quantity that is the result of evaluating an expression that includes an undefined label, it lists a '*' for each digit of the quantity printed (e.g., an unresolved hex byte is listed as '**', while an unresolved octal word appears as '******'). If the .PATCHLIST directive is used, the assembler lists patch messages every time it encounters a label declaration that enables it to resolve all occurrences of a forward reference to that label. The messages (one for every backpatch performed) appear before the source statement that contains the label in question, and are of the form:

<location in codefile patched>* <patch value>

With this feature, the listing describes the contents of each byte or word of emitted code; if neatness of the assembled listing is more desirable, the .NOPATCHLIST directive will suppress the patch messages.

VI.7.3.2 External References

When the assembler emits a word quantity that is the result of evaluating an expression that contains an externally referenced label, the value of that label (which cannot be determined until link time) is taken as zero; therefore, the emitted value will reflect only the result of any assembly time constants that were present in the expression.

VI.7.3.3 Multiple Code Lines

Sometimes, it is possible for one source statement to generate more code than will fit in the code field; in most cases, the code is listed on successive lines of the code field (with corresponding blank source listing fields). Three exceptions are the .ORG, .ALIGN, and .BLOCK directives; because most uses of these directives

generate large numbers of uninteresting byte values, the code field for these arguments is limited to as many bytes as will fit in the code field of one line.

VI.7.4 Symbol Table

The symbol table is an alphabetically sorted table of entries for all symbols declared in the source program. Each entry consists of three fields; the symbol identifier, the symbol type, and the value assigned to that symbol. The symbol identifiers are defined in a dictionary printed at the top of the symbol table. Symbols equated to constants have their constant values in the third field, while program labels are matched with their location counter offsets; all other symbols have dashes in their value field, as they possess no values relevant to the listing.

VI.7.5 Example

The following is a small example of an assembled listing and the text of the corresponding host program:

```
PROGRAM EXAMPLE;      (* Pascal host program *)
const size = 80;
var i,j,k : integer;
     lst1 : array [0..9] of char;
     (* PRT and LST2 get allocated here *)

begin
    k := 45;
    do_nothing;
    j := null_func(k,size);
end.
```


```
000000¦
000000¦                          .RELPROC DONOTHING     ;underscores are not significant
000000¦                                                 ;in PASCAL
000000¦                          .CONST  SIZE           ;can get at size constant in host...
000000¦                          .PUBLIC I,LST1         ;and also these two global vars
000000¦                          .DEF    TEMP1          ;this allows NULLFUNC to get at temp1
000000¦                                                 ;code starts here...
000000¦ 000000              TEMP1  .WORD
000002¦ 000000              RETADR .EQU TEMP1
000002¦ 120 261 000000              LDMD    R20,RETADR  ;reference a memory location
000006¦
000006¦                          ; does nothing
000006¦
000006¦ 211                      ICM     R20            ;increment it
000007¦ 263 000000               STMD    R20,RETADR     ;and store it
000012¦ 236                      RTN
000013¦
000013¦                                                 ;end of procedure DONOTHING
000013¦
000013¦


AB - Absolute     LB - Label     UD - Undefined     MC - Macro
RF   Ref          DF - Def       PR - Proc          FC - Func
PB - Public       PV - Private   CS - Consts


DONOTHIN PR ------¦ I     PB ------¦ LST1   PB ------¦ RETADR  LB 000000¦ SIZE    LB ------¦ TEMP1  DF 000000
```

```
000000¦                          .RELFUNC NULLFUNC,2     ;start function of 2 parameters
000000¦
000000¦                          .PRIVATE PRT,LST2:9.    ;10 words of private data
000000¦                          .REF     TEMP1          ;references data TEMP in DONOTHING
000000¦                                                  ;code starts here
000000¦
000000¦ 120 012 343              POMD     R20,-R12       ;get parameter 'z' from stack
000003¦ 263 000000               STMD     R20,PRT        ;store it
000006¦ 343                      POMD     R20,-R12       ;get parameter 'xxyxx'
000007¦ 263 000004               STMD     R20,LST2+4     ;store it too
000012¦ 343                      POMD     R20,-R12       ;toss 1 word of junk
000013¦
000013¦                          ; performs null action
000013¦
000013¦ 345                      PUMD     R20,+R12       ;return xxyxx as result
000014¦ 236                      RTN                     ;data starts here...
000015¦
000015¦                          .END                    ;end of assembly


AB - Absolute     LB - Label     UD - Undefined     MC - Macro
RF   Ref          DF - Def       PR - Proc          FC - Func
PB - Public       PV - Private   CS - Consts


LST2     PV ------¦  NULLFUNC FC ------¦  PRT      PV ------¦  TEMP1    RF ------¦


Assembly complete:        38 lines
     0   errors flagged on this assembly
```

VI.8 HP-86/87 Assembler

VI.8.1 Syntax Conventions

The HP-86/87 assembler adheres to HP standard syntax for opcode fields, register names, and address modes with the following exceptions:

The register number alone (without the "R" prefix) may also be coded (except as the address register of a push or pop instruction).

The symbol "*R" is used to denote indirect loading of the register pointer (as opposed to R*).

The symbol "$" is used to denote the current value of the location counter in an expression.

The GTO pseudo instruction is built into the assembler. It should be used rather than the JMP instruction, when the relative distance exceeds -128 or +127. Example:

```
    GTO     LOCATION
```

is assembled as:

```
    DRP     PC
    LDM     PC,=LOCATION-1
```

The standard syntax for the Literal Immediate, Literal Direct, and Literal Indirect addressing modes requires the literal or label to be preceded by the "=" character. The HP-86/87 assembler is not particular about this except in the case of the Literal Immediate addressing mode, for example,

```
    LDM     R45,=111,222,333
```

In other cases the the assembler is expecting a two-byte address, either absolute or relocatable. Moreover, a superfluous "=" preceding an address in an indexed mode instruction is ignored.

Indexed mode instructions may not specify the index register as "Xnn," instead, the standard name "Rnn" should be used.

EXAMPLES:

```
LDMD    R40,R20,=ARRAY      ;R20 is the index register
JSB     R14,=SUBROUTINE     ;R14 is the index register
STMD    R40,X20,=ARRAY      ;will cause an assembly error
```

Note that it is possible to define a label Xnn and use the label in a register field. The assembler will not restrict the label to an index register field however.

EXAMPLE:

```
X20     .EQU  R20
        LDMD R40,X20,=ARRAY  ; R20 is the index register
```

VI.8.2 Predefined Constants

The register names, R0 through R77, are predefined. In addition, the symbols PC (program counter) and SP (stack pointer) are predefined as R4 and R6 respectively.

EXAMPLE:

```
LDM     PC,=LABEL           ;equivalent to: LDM R4,=LABEL
LDM     SP,=102000          ;equivalent to: LDM R6,=102000
STMD  R10,R12
LDMD  R40,R20,=TABLE
```

VI.8.3 Additional Assembler Directives

.ARP — Does not generate any code but allows the programmer to assert the validity and contents of the ARP and DRP. A simple integer in the operand field specifies the contents of the ARP or DRP; no operand specifies that the ARP or DRP is invalid or unknown.

FORM:
```
[b] .ARP [integer]
[b] .DRP [integer]
```

EXAMPLE:

```
        .ARP 40             ;specify that ARP contains 40
        .DRP 6              ;specify that DRP contains 6
        .ARP
```

The assembler maintains records of the data and address register numbers specified by the last instruction assembled, and validity indications for each. ARP and DRP instructions are therefore emitted only when the corresponding register is changed, or has become invalid.

The following usages cause the remembered register pointer value to become invalid:

1. The start of a .PROC, .FUNC, .RELPROC or .RELFUNC invalidates both ARP and DRP.
2. The directives .ASECT, .PSECT, .ORG, and .BLOCK invalidate both ARP and DRP.
3. A JSB (Jump to Subroutine) invalidates both ARP and DRP.
4. A label in the label field of a source line invalidates both ARP and DRP.
5. The coding of R# for either data register or address register (or both) prevents the generation of the corresponding DRP or ARP, as in the standard HP conventions; however, the corresponding register record becomes invalid for subsequent instructions.

VI.8.4 Sharing of Machine Resources with Interpreter

The return address is on the subroutine return stack. All registers are available for use in the assembly routine with the exception of R12-13 (evaluation stack pointer) and R16-17. Extended Memory Controller registers PTR1 and PTR2 are available for use in the assembly routine.

The evaluation stack is an increasing stack and contains any parameters passed to the assembly routine. R12 points to the top of the evaluation stack. Parameters must be removed from the evaluation stack and function result is returned on the evaluation stack. The stack is 640 bytes deep and may be used by the assembly routine for temporary storage.

Pointers passed as parameters are 16 bit pointers with a pointer value of 000000 defined as the first byte of extended memory space, or pointers may be thought of as relative to octal address 200000.

VI.8.5 Memory Organization

The HP-86/87 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first. In the representation of a 16-bit memory

address, a binary integer, a BCD integer, or a floating point number, the most significant byte is at the numerically largest address.

The first 24k bytes of memory are system ROM. The next 8k bytes, octal addresses from 060000 through 077777, are allocated to plug-in ROM. System RAM memory extends from 100000 through 177377. Memory-mapped I/O addresses are in the range 177400 through 177777. In addition, an extended RAM memory module with 64k bytes is addressed from 200000 through 277777, 96k extended RAM is addressed from 200000 through 477777, and 128k of extended RAM is addressed from 200000 through 577777.

VI.8.6 Default Constant and List Radices

The default constant radix is octal and the listing radix is octal. Decimal integers are denoted by the suffix ".",e.g., 64. is 100 octal. Binary coded decimal (BCD) constants must be specified as hexadecimal constants. using the suffix "H", e.g., 64H = 100. = 144 octal. If the .RADIX directive is used to specify a different default radix, the suffix "Q" denotes an octal constant.

VI.8.7 Additional Assembler Error Messages

In addition to the normal assembler error messages, the following messages are specific to the HP-86/87 assembler:

76: ARP/DRP operand not register # 0 to 63
77: Address expected
78: Register 1 named (warning)
79: Literal expected after '='
80: Invalid expression for data register
81: Missing first (data register) operand
82: Missing second operand
83: Invalid expression for address register
84: Comma expected between operands
85: Invalid second operand address mode
86: # immediate bytes doesn't agree with dr
(also given if multiple bytes with byte mode instruction)
87: # immediate bytes not verified
(R# or *R used in data register field)
88: Missing jump address

VII. SEGMENTS, UNITS, AND LINKING

VII.1 Overview

Segments, units, and linking are three major facilities which help the user manage program files and the use of main memory. These facilities permit the development of very large programs in a microsystem environment, and in fact have been used extensively in the development of the System itself.

The techniques offered by the System fall broadly into two categories: run-time main memory management, and separate compilation.

VII.1.1 Main Memory Management

Not all of a program need be in main memory at runtime. Most programs can be described in terms of a "working-set" of code which is required over a given period of time. For most (if not all) of a program's execution time, the working-set is a subset of the entire program -- sometimes a very small one. Portions of a program which are not part of the working-set can reside on disk, thus freeing main memory for other uses.

When the p-System executes a codefile, it reads code into main memory and runs it. When the code has finished running, or the space it occupies is needed for some action of higher priority, the space it occupies may be overwritten with new code or new data. Code is "swapped" into main memory a segment at a time.

In its simplest form, a Pascal segment includes a main program and all of its routines. A routine may occupy a segment of its own: this is accomplished by declaring it a SEGMENT routine. SEGMENT routines may be swapped independently of the main program; declaring a routine to be a SEGMENT is a useful means of managing the use of main memory.

Routines which are not part of a program's main working-set are prime candidates for occupying their own segment. Such routines include initialization and wrap-up procedures, and routines that are used only once or only rarely while a program is executing.

Reading a procedure in from disk before it is executed does take time, and so the selection of which procedures to make disk-resident should be done judiciously.

VII.1.2 Separate Compilation

Separate compilation, also referred to as "external compilation", is a technique whereby portions of a program are compiled separately from each other, and subsequently executed as a co-ordinated whole.

Many programs are too large to compile within the memory confines of a particular microcomputer. Such programs might comfortably run on the same machine, especially if they are segmented as described above. The Operating System is a case in point. Compiling small pieces of a program separately is the way to overcome such a memory problem.

Separate compilation also has the advantage of allowing only small portions of a program to be changed without affecting the rest of the code. This saves much time and is less error prone. Libraries of correct routines may be built up and used in the development of other programs. This capability is important if a large program is being developed, and invaluable if the project involves several programmers.

These considerations also apply to assembly language programs. Large assembly programs (such as P-machine emulators) can often be more effectively maintained in several separate pieces. When all these pieces have been assembled, a "link editor" (the System's Linker) stitches them together by installing the linkages that allow the various pieces to reference each other and function as a unified whole.

It may also be desirable to reference an assembly language routine from a higher-level language host program (e.g., Pascal or FORTRAN). This may be necessary for performance reasons, or to provide low-level machine-dependent or device-dependent handling.

The p-System allows assembly language routines to be linked in with other assembly routines, or into higher-level hosts (programs or units). Refer to Chapter VI on the Adaptable Assembler.

In UCSD Pascal, separate compilation is achieved by the UNIT construct. A UNIT is a group of routines and data structures. The contents of a UNIT usually relate to some common application, such as screen control or datafile handling. A program or another UNIT (called a "client module" or "host") may use the routines and data structures of a UNIT by simply naming it in a 'USES' declaration. A unit consists of two main parts: the INTERFACE part, which can declare constants, types, variables, procedures, processes, and functions that are public (available to any client module), and the IMPLEMENTATION part, in which private declarations can be made. These private declarations are available only within the UNIT, and not to client modules. Units can either be embedded in a host, or compiled separately.

The code for a UNIT that is used by a program may reside in *SYSTEM.LIBRARY, or in another codefile. If it is in another codefile, the programmer may inform the Compiler of this by using the $U compile-time option, and inform the Operating System by including the codefile's name in a "library text file." The default library text file is *USERLIB.TEXT, but can be changed by an execution option. See Sections VII.3 and II.3.

The FORTRAN Compiler has the $USES directive for separately compiled units; refer to the FORTRAN Reference Manual for details.

VII.1.3 General Tactics

This section offers some advice on the use of SEGMENTs and UNITs. It presents a scenario for the design of a large program, with some strategies that might be used. UNITs and SEGMENTs are useful means of decomposing large programs into independent tasks.

On microprocessor systems, the main bottlenecks in the development of large programs are: (1) a large number of variable declarations that consume space while a program is compiling, and (2) large pieces of code using up memory space while the program is executing. UNITs address the first problem by allowing separate compilation, and minimizing the number of variables that are needed to communicate between separate tasks. SEGMENTs address the second problem by allowing only code that is in use to be present in main memory (while unused code is disk-resident) at any given time.

A program can be written with runtime memory management and separate compilations already planned, or it can written as a whole and then tuned to fit a particular system. The latter approach is feasible when one is unsure about the necessity of using SEGMENTs, or is quite sure that they will be used only rarely. The former approach is preferred, and is usually less painful to accomplish.

A typical scenario for the construction of a relatively large application program might be as follows:

1) Design the program (user and machine interfaces).

2) Determine needed additions to the library of utilities
   -- both general and applied tools.

3) Write and debug utilities, and add to libraries.

4) Code and debug the program.

5) Tune the program for better performance.

During the design, one should try as much as possible to use existing procedures, so as to decrease coding time and increase reliability. This strategy can be assisted by the use of UNITs.

To determine segmentation, the programmer should consider the expected execution sequence, and attempt to group routines inside SEGMENTs so that the SEGMENT routines are called as infrequently as possible.

It is also important that SEGMENT routines be independent. They should not call

routines in different segments (including non-SEGMENT routines); if they do, then both segments must be in memory at the same time: this eliminates the advantage of segmentation.

While designing the program, one should also consider the logical (functional) grouping of procedures into UNITs. As well as making the compilation of a large program possible, this can aid the program's conceptual design (and therefore the testing of it). UNITs may contain SEGMENT routines, so the two techniques may be combined.

The programmer should be aware that a UNIT occupies a segment of its own (except possibly for any SEGMENT routines it may contain). The UNIT's segment, like other code segments, remains disk-resident except when its routines are being called.

Steps (2) and (3) are aimed at capturing some of the new routines in a form which will allow them to be used in future programs. At this point the design should be reviewed (and perhaps modified) with the objective of identifying those routines which might be useful in the future. Needed routines might be made somewhat more general, and put into libraries.

It is usually a good practice to program and test such utilities before moving on to programming the remainder of the program. Doing so tends to ensure that more generally useful procedures are added to the library, since it helps one avoid the tendency to tailor them to the particular program being developed.

The INTERFACE part of a UNIT should be completed before the IMPLEMENTATION part, especially if several programmers are working on the same project.

Tuning a program usually means performance tuning. Since SEGMENTs offer greater memory space at reduced speed, it may be that performance is improved by turning routines into SEGMENT routines, or by turning SEGMENT routines back into normal routines. Either route is feasible. Some attention must be paid to the rules for declaring SEGMENTs: see the next section of this chapter.

Sections VII.2 and VII.3 of this chapter describe the syntax of using UNITs and SEGMENT routines in Pascal. For information on other languages, refer to the appropriate manual.

VII.2 Segments

The declaration of a segment routine is no different from other routine declarations (i.e., procedures, functions, and processes), except that it is preceded by the UCSD reserved word 'SEGMENT'.

For example:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
   { Pascal code here }
END;
```

Declaring a routine as a segment routine does not change the meaning of the Pascal program, but affects the time and space requirements of the program's execution. The segment routine and all of its nested routines (except a nested routine that is itself a segment routine) are grouped together in what is called a "code segment".

A program and its routines are all compiled as a single code segment, unless some routines have been declared as SEGMENTs. Since a code segment is disk-resident until it is used, and since the space it occupies in memory may be overwritten when it terminates, declaring once-used or little-used routines as SEGMENTs may improve a program's utilization of main memory.

Up to 255 segments may be contained within one program. The "bodies" (that is, the BEGIN-END blocks) of all segment routines must be declared before the bodies of all non-segment routines within a given code segment. This applies to both segment routines and main programs. If a segment routine calls a non-segment routine, the non-segment routine must be forward-declared, because its body cannot precede the body of any segment routine (including its caller).

No SEGMENT routines may be declared in the INTERFACE section of a UNIT; they may be declared in the IMPLEMENTATION section.

No EXTERNAL routine may be a SEGMENT routine.

Outside of these restrictions, any routine may be declared a SEGMENT.

Example:

```
PROGRAM GOLE;

   SEGMENT PROCEDURE STRENGAL;
      ...
      BEGIN
      ...
      END;

   PROCEDURE MYNDAL (FLAK: INTEGER); FORWARD;

      { MYNDAL is not a SEGMENT routine, and
        therefore must be declared FORWARD  }

   SEGMENT FUNCTION MOAD (PART,WHOLE: REAL): INTEGER;
      ...
      BEGIN
      ...
      END;

   PROCEDURE MYNDAL;
      ...
      PROCEDURE EARLY (1: UNREAL);
         ...
         SEGMENT PROCEDURE LATE (J: IMAGINARY);
            ...
            BEGIN
               { note that this may be a segment:
                 it precedes all code bodies within
                 the enclosing code segment
                 (i.e., GOLE)                        }
            END {LATE};

         BEGIN
         ...
         END {EARLY};

      BEGIN
      ...
      END {MYNDAL};

BEGIN
   ...
END {GOLE}.
```

VII.3 Units

A UNIT is a group of interdependent procedures, functions, processes, and associated data structures, which are usually related to a common area of application. Whenever a UNIT is needed within a program, the program declares it in a USES statement. A UNIT consists of two main parts: an INTERFACE part, which declares constants, types, variables, procedures, functions, and processes that are public and can be used by the host (program or other UNIT), and an IMPLEMENTATION part, which declares labels, constants, types, variables, procedures, functions, and processes that are private, not available to the host, and used only within the UNIT. The INTERFACE part declares how the program will communicate with the user of the UNIT, while the IMPLEMENTATION part defines how the UNIT will accomplish its task.

The syntax of a UNIT may be sketched as follows:

```
UNIT <unit identifier>;

INTERFACE
    USES <unit identifier list>;
    <constant definitions>;
    <type definitions>;
    <variable declarations>;
    <routine headings>;

IMPLEMENTATION
    USES <unit identifier list>;
    <label declarations>;
    <constant definitions>;
    <type definitions>;
    <variable declarations>;
    <routine declarations>;

[ BEGIN
     <initialization statements>
     ***;
     <termination statements>]
END
```

The INTERFACE part may only contain routine headings -- no bodies. The bodies of routines declared in the INTERFACE part are fully defined in the IMPLEMENTATION part, much as FORWARD procedures are fully defined apart from their original declaration.

An INTERFACE part is terminated by the UCSD reserved word IMPLEMENTATION.

An INTERFACE part may not contain $Include files (see Pascal manual). An INTERFACE part may be contained within an $Include file, provided that all of the INTERFACE is in the $Include file; i.e., an INTERFACE part may not cross an $Include file boundary. Note that IMPLEMENTATION terminates an INTERFACE part, so that if an INTERFACE part is contained in an $Include file, the $Include file must contain both the reserved words INTERFACE and IMPLEMENTATION.

Example:

```
UNIT GOLE1;
INTERFACE
   {$I INTER_DECS}
IMPLEMENTATION
   ...
END;
```

```
UNIT GOLE2;
{$I INTER_PART}
IMPLEMENTATION
...
END;
```

... are not legal forms of a UNIT, while the following outline is:

```
UNIT GOLE3;
{$I WHOLE_UNIT}
;
```

The <initialization statements> and <termination statements> are optional sections of code. Initialization statements, if present, are executed before any of the code in a host that USES the UNIT is executed, and termination statements, if present, are executed after the host's code has terminated.

Initialization statements are separated from termination statements by the line '***;'. Either the section of initialization statements, or the section of termination statements, or both, may be empty.

The construct '***;' is adapted from the Pascal dialect called Pascal Plus: see "Pascal Plus -- Another Language for Modular Multiprogramming," by J. Welsh and D.W. Bustard, in "Software -- Practice and Experience," Vol. 9, No. 11, November, 1979, pp. 947-957. In Pascal Plus, '***' has the full status of a statement, while in UCSD Pascal, '***;' may only be used to separate initialization code from termination code within the statement section of a UNIT.

Example:

The following are all legal code bodies of a UNIT:

```
END {there is no initialization or termination code};


BEGIN
   {this is initialization code}
   INIT_ARRAYS;
   FLAG := FALSE;
   COUNT := 23;
   ***;
   {this is termination code}
   SEMINIT ( LIGHT, 0 );
END {UNIT};


BEGIN
   ***;
   {this is all termination code}
   INIT_ARRAYS;
   FLAG := FALSE;
   COUNT := 23;
   SEMINIT ( LIGHT, 0 )
END {UNIT};


BEGIN
   {this is all initialization code}
   INIT_ARRAYS;
   FLAG := FALSE;
   COUNT := 23;
   SEMINIT ( LIGHT, 0 )
END {UNIT};
```

The statement part of a UNIT should not contain GOTO statements which branch around the '***;' separator: the effect of executing such statements is not fully predictable.

A UNIT's statement part may contain statements of the form: EXIT(PROGRAM) (EXIT(<unitname>) is not allowed). An EXIT(PROGRAM) in the initialization code has the effect of skipping the remainder of the initialization code (if any) and the host's code: execution proceeds with the UNIT's termination section. An EXIT(PROGRAM) in the termination code skips the remainder of the termination code (there may be termination code from other hosts still waiting to execute -- the EXIT does not abort the execution of these other termination sections).

To use one or more UNITs, a program must name them in a USES declaration immediately following the program heading (before the <block>). Upon encountering a USES declaration, the Compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all identifiers declared in the INTERFACE part are global. Name conflicts may arise if the host defines an identifier already defined in the UNIT.

A UNIT may also USE another UNIT. In this case, the USES declaration may appear at the beginning of either the INTERFACE part or the IMPLEMENTATION part. Since USES may be nested, if they appear in the INTERFACE part, the ordering of a USES declaration may be important: if UNIT_A USES UNIT_B, then the host must specify that it USES UNIT_B before it USES UNIT_A.

Routines declared in the INTERFACE part must not be SEGMENT routines, but SEGMENT routines can be declared in the IMPLEMENTATION part. (Declaring SEGMENTs within UNITs is subject to the same ordering as within a main program; see above, Section VII.2.)

For purposes of listing a program, the Compiler treats a INTERFACE section as an include level. Thus, $Include file nesting is restricted within the scope of a USES declaration.

The UCSD System will compile a Pascal program, a single UNIT, or a string of UNITs (separated by semicolons). A Pascal program may define a UNIT in-line. An in-line UNIT definition must appear between the program heading and the <block>. This has the advantage of simplicity, but if changes are made to either the program or the UNIT, both must be recompiled.

UNITs need not be explicitly linked together. At compile-time a USEd UNIT's INTERFACE part must be referenced by the Compiler. If the UNIT's source is in the host program's source, or if the UNIT's code is in *SYSTEM.LIBRARY, nothing more need be specified. If the UNIT's code resides in a different file (a "user library"), the $U Compiler directive must be used to specify which file.

At runtime, the code (all code, in fact) must be in either the user program, *SYSTEM.LIBRARY, a user library, or the Operating System. If a unit is in a user library, the name of the library file must appear in a "library text file." To find a UNIT's code, the System searches first the files named in a library text file (in order), and then *SYSTEM.LIBRARY. If no library text file is present, the System searches *SYSTEM.LIBRARY alone. The default library text file is called *USERLIB.TEXT; this default may be changed by an execution option (see Section II.3).

Example:

The following might be the contents of a library text file:

```
FUN:ADVENT.LIB
curve
tg: graphics
PLAY
```

... for each UNIT encountered in the host, the System searches first ADVENT.LIB (which must reside on the volume FUN:), then CURVE.CODE (which must reside on the default volume), and so forth. Failing to find a UNIT in these four files, the System searches *SYSTEM.LIBRARY.

As indicated in the example, specifying the .CODE suffix to a filename is optional in the library text file's list.

The name *SYSTEM.LIBRARY may be included in a library text file. If this is the case, it is searched on order, as it appears.

Changes in a host program require only that the user recompile the program. Changes in the IMPLEMENTATION part of a UNIT only require the user to recompile the UNIT. Changes in the INTERFACE part of a UNIT require that the user recompile both the UNIT and all hosts that USE that UNIT.

Earlier versions of UCSD Pascal had several varieties of UNITs with different internal implementations. Declarations for these old UNIT types (SEPARATE and INTRINSIC) are still accepted by the Compiler, but now the same implementation is used for all.

The use of UNIT-style mechanisms in the System's other high-level languages is discussed in the documentation for each particular language. External linkages involving assembled routines are discussed in Chapter VI and in the next section.

VII.4 The Linker

The Linker is a System program (accessed by the L(ink command at the System level) which allows EXTERNAL code to be linked in to a Pascal or FORTRAN program. EXTERNAL routines are routines (Procedures, functions, or processes) that are written in an assembly language and conform to the p-System's calling and parameter-passing protocols. They are declared EXTERNAL in the host program, and must be linked before the program is run. The Linker may also be used to link together separately assembled pieces of a single assembly program.

The Linker is a program of the sort called a "link editor". It stitches code together by installing the internal linkages that allow various pieces to function as a unified whole.

When a program which must be linked is R(un, the Linker will automatically search *SYSTEM.LIBRARY for the necessary external routines. In all other cases (i.e., the user used eX(ecute instead of R(un, or the library is not SYSTEM.LIBRARY), the user is responsible for "manually" linking the code before executing it.

When the Linker is called automatically and cannot find the needed code in *SYSTEM.LIBRARY, it will respond with an error message:

```
      Proc,
      Func,
      Global,
  or Public <identifier> undefined
```

To link code "by hand", call the Linker by typing 'L' at the command level.

VII.4.1 Using the Linker

The Linker prompts for several filenames, and as it reads and links code together, displays the names of what it is linking. The prompts are, in order:

Host file?

... the hostfile is an existing code file into which the external routines are to be linked. Filename conventions apply here (.CODE is automatically appended to all filenames except '*<return>' or any filename that ends in a '.'). The response '*<return>' or simply <return> causes the Linker to open *SYSTEM.WRK.CODE. The Linker then asks for the names of library files in which the external routines are to be found:

Lib file?

... any number of library files may be specified. The prompt will keep reappearing until <return> is typed. Responding '*<return>' opens *SYSTEM.LIBRARY. The success of opening each library file is reported.

Example (underlined portions are user input):

```
Lib file? *<return>
Opening *SYSTEM.LIBRARY
Lib file? FIX.8<return>
No file FIX.8.CODE
Type <sp>(continue), <esc>(terminate)
Lib file? FIX.9<return>
Opening FIX.9.CODE
bad seg name
Type <sp>(continue), <esc>(terminate)
Lib file?

... and so forth
```

When the names of all library files have been entered, the Linker prompts for:

Map name?

This is the name of an existing text file that the Linker will use for detailing the linkage of the external routines into the host codefile, including the base addresses and lengths of the routines. If the response is just <return>, no map file will be created.

After the 'Map name?' response, the Linker reads all the necessary routines from the codefiles. It then asks for the destination file for the linked code output:

Output file?

... this is a codefile name (often the same as the host file). The .CODE suffix must be included. If the user types just <return>, output will be to the workfile (*SYSTEM.WRK.CODE).

After this last prompt, the Linker commences actual linking. During linking, the Linker displays the names of all routines being linked. A missing or undefined routine causes the Linker to abort with the '<identifier> undefined' message described above.

If linking is successful, the user has a unified codefile that may be eX(ecuted.

The user should note that, since the files may be assembled files, they may be of either byte sex. All files linked together must be of the same byte sex, but the Linker will produce a correct codefile regardless of which byte sex that is, or whether it is the same as the machine on which the Linker is running. More information on byte sex is given in the Installation Guide.

The codefile produced by the Linker contains routines in the order in which they were given as contained in the library files. This is important to note if the program is an all-assembly file. The codefile contains first routines from the host file, and then library file routines, all in their original order.

The next section contains more information on libraries.

VII.5 The Utility LIBRARY

LIBRARY.CODE is a utility program that allows the user to group separate compilatons (UNITs or programs) and separately assembled routines into a single file. A library is a concatenation of such compilations and routines. Libraries are a useful means of grouping the separate pieces needed by a program or group of programs. Manipulating a single library file takes less time than if the various pieces it contains were each within an individual file. Libraries generally contain routines relating to a certain area of application; they can be used for functional groupings much as UNITs can. Thus, a user might want to maintain a math library, a datafile-management library, and so forth -- each of these libraries containing routines general enough to be used by many programs over a long period of time.

Individual programs might also take advantage of the library construct. If a program uses several UNITs suitable for compiling separately, but the UNITs themselves are too small to warrant putting each into its own file, the user would want to construct a single library containing all of those UNITs.

Even if a file contains only a single UNIT or routine, it is treated as a library when the UNIT or routine is used by some external host.

LIBRARY is useful for putting UNITs into SYSTEM.LIBRARY or other libraries, grouping assembly routines together, and so forth.

This section uses the term "compilation unit". A program or UNIT and all the SEGMENTs declared inside it are called a compilation unit. The SEGMENT for the program or UNIT is called the host segment of the compilation unit. SEGMENT routines declared inside the host are called subsidiary segments. UNITs used by the host are <u>not</u> considered to be segments belonging to that compilation unit. UNITs used by the compilation unit generate information in the host segment called segment references ("seg refs" for short). The seg refs contain the names of all segments referenced by a compilation unit, and the Operating System uses this information to set up a runtime environment.

Some routines called from hosts exist in UNITs in the Operating System, and therefore appear in seg refs, even though there is no explicit USES declaration. For example, WRITELN resides in the Operating System UNIT PASCALIO, so the name PASCALIO will appear in the seg refs of any host that calls WRITELN.

VII.5.1 Using LIBRARY

When LIBRARY is executed, a prompt asks for an output filename. The filename must end in .CODE. LIBRARY will <u>remove</u> an old file with the same name as the new library.

LIBRARY then prompts for the input filename. .CODE is automatically appended.

EXAMPLE:

The user specifies SCREENOPS.CODE as an input file. LIBRARY displays the following:

```
Library: N(ew, 0-9(slot-to-slot, E(very, S(elect, C(omp-unit, F(ill,?

Input file? SCREENOPS<return>
  0 u SCREENOP   921                8
  1 s SEGSCINI   416                9
  2                                10
  3                                11
  4                                12
  5                                13
  6                                14
  7                                15

Write to what file? NEW.CODE<return>
  0                                8
  1                                9
  2                                10
  3                                11
  4                                12
  5                                13
  6                                14
  7                                15
```

... the display shows that the file SCREENOPS consists of a UNIT and a SEGMENT routine. There are four possible types of code that can occupy the 16 "slots" in a library: **u**nits, **p**rograms, **s**egment routines, and **a**ssembled routines. LIBRARY displays the type, along with the name and length (in words) of each module.

LIBRARY's promptline shows the various commands available.

N(ew prompts for a new input file.

A(bort stops LIBRARY without saving the output file.

Q(uit stops LIBRARY and does save the output file. When the user Q(uit's LIBRARY, it prompts 'Notice?' at the bottom of the screen. A copyright notice

to be placed in the output file's segment dictionary may be typed in (followed by <return>). Simply typing <return> exits LIBRARY without writing a copyright notice.

T(og toggles a switch which determines whether or not INTERFACE parts of UNITs are copied to the output file.

R(efs lists the names of each entry in the segment reference lists of all segments currently in the output file. The list of names also includes the names of all compilation units currently in the output file, even though their names may not occur in any of the segment references.

The remaining five commands allow code segments to be transferred from the input file to the output file.

A given "slot" can be transferred to the output file by typing a digit (0..9). LIBRARY then prompts: 'Copy from slot # ?' and displays the digit just typed. If that is the name of the slot, type <space>. If that is the first digit of a two-digit slot number, type in the second digit and follow it with a <space>. LIBRARY confirms your entry before actually copying code. <backspace> may be used to correct errors. If <return> is typed when no number is shown, the copy does not happen and LIBRARY's promptline is redisplayed.

If the destination slot in the output file is already filled, a warning says so and no copy takes place. If an identical code segment is already present anywhere in the output file, the new code segment is copied anyway.

E(very causes all of the code in the input file to be copied to the output file. If, for any code segment, the corresponding slot in the output file is alread filled, then LIBRARY searches for the next available slot and places the code there. If, for any code segment, an identical code segment already exists in the output file, that segment is not copied over.

S(elect causes LIBRARY to prompt the user for which code segments to transfer. For each code segment not already in the output file, LIBRARY prompts: 'Copy from slot #_?'. A 'Y' or 'N' causes the segment to be copied or passed by, an 'E' causes the remainder of the code segments to be transferred (as in E(very), a <space> or <return> aborts the S(elect. If the corresponding slot in the output file is filled, LIBRARY searches for the next available slot and places the code there.

C(omp-unit causes LIBRARY to prompt: 'Copy what compilation unit?'. The compilation unit named is transferred along with any segment procedures that it references. Procedures already present in the output file are not copied.

F(ill does the equivalent of a C(omp-unit command for all the compilation units referenced by the segment references in the output file.

VIII. CONCURRENT PROCESSES

UCSD Pascal allows the user to declare and initiate concurrent processes. A process is a procedure whose execution appears to proceed at the same time as (i.e., concurrently with) the main program. Processes are declared as procedures are declared, and set into action by the intrinsic START. Thus, more than one process may run at once, and the same process may be START'ed several times.

On machines which have only one processor (this includes the vast majority of UCSD Pascal installations) the System shares the (physical) processor between various (Pascal) processes. This switching may lead to an overall increase in program execution time. Processes are nonetheless useful in a variety of applications, particularly interrupt handling.

VIII.1 Introduction

A process is declared exactly as a procedure would be, with the (UCSD) reserved word PROCESS replacing the reserved word PROCEDURE.

Examples:

```
PROCESS ZIP;
   BEGIN ... END;

PROCESS DINNER (var SPLIT, BLACKEYED : peas);
   begin ... end;
```

A process is started by the UCSD intrinsic START. The principal parameter passed to START is a call to a process, e.g., START(ZIP) or START(DINNER(7,234)). START also takes three optional parameters, which are explained after the following example:

```
PROGRAM DUFFER;
          var PID : processid;
              I, J : integer;

          PROCESS BLUE;
            begin
              .
              .
            end;

          PROCESS RED ( X, Y : integer );
            begin
              .
              .
            end;

          begin
            start( BLUE );
            I := 1;  J := 2;
            start( RED( I, J ) );
            start( RED( 3, 4 ), PID );
            start( RED( 5, 5 ), PID, 300 );
            start( RED( J, I ), PID, I+J, 10 );
            .
            .
          end.
```

In the example above, program DUFFER starts processes RED and BLUE. In fact, RED is started several times. The five processes started will each run to completion, as will the main program, and the (physical) processor will share time among them. Note that the four invocations of RED result in four different versions of RED being started, each (in this example) with different parameter values.

Each invocation of a process is assigned an internal PROCESSID. PROCESSID is a UCSD predeclared type. The user may learn what processid has been assigned a given process invocation by using an optional second parameter. Thus, in START(RED(3,4), PID); the variable PID is set to a new PROCESSID value. Processids are chiefly for the use of the System and system programmers.

The optional third parameter to START is the stacksize parameter. It determines how much memory space is allocated to the process invocation (the default is 200 words).

The optional fourth parameter to START is a priority value. This determines the proportion of processor time that the process will receive before it is completed. The priorities assigned to processes are used by the System to decide which active process gets to use the available processor. Higher priority processes are given the processer more often than lower priority processes. If no priority value is given in START, the new process inherits the priority value of its caller.

VIII.2 Semaphores

Semaphores may be used in two basic ways:

(1) for mutual exclusion problems: controlling access to "critical sections" of code;

(2) for synchronization between "cooperating processes".

An extremely common application employing both of these capabilities is resource allocation. In UCSD Pascal it is also possible to associate semaphores with hardware interrupts and use them to write interrupt handlers in Pascal. We shall discuss these uses below.

The name "semaphore" was coined by E. W. Dijkstra as an analogy to a railroad traffic signal. The railroad semaphore controls whether or not a train may enter the next section of track; a train passing the semaphore when it is "green" automatically switches it to "red", preventing further trains from entering that section of track until the privileged train has exited, at which time the semaphore is switched to "green" again.

Semaphores themselves may be divided into two classes: Boolean and counting semaphores. A semaphore which has only two states (e.g., red and green) is referred to as a Boolean semaphore. If more than two states are allowed, it is called a counting semaphore. In UCSD Pascal, counting semaphores may span the range [0..maxint]. The zero is analogous to the "red" or stop value. It is possible to use counting semaphores as Boolean semaphores if one is careful to restrict oneself to only the values 0 and 1.

Given a set of concurrent processes and a single semaphore variable which they test, we can imagine that each process (or "train") is running on a private processor ("track") with separate indicators of the semaphore value under some central control. For example, there may be a section of track which must be shared by all the trains, but only a single train is to be allowed in that section at a time. When the value of the semaphore is zero, the central control will cause any trains that approach the semaphore to stop and wait until they are individually signalled to proceed. When the central control determines that it is safe for a train to continue (i.e. when some other train has left the common section of track) it will select <u>one</u> (only) of the trains waiting and signal it to go on.

The UCSD intrinsics which manipulate semaphores are SEMINIT, WAIT, SIGNAL, and ATTACH.

SEMINIT initializes a semaphore by assigning it a count and an empty queue. All semaphores must be initialized in this way, or their value (and hence the results of

a program!) is unpredictable.

WAIT causes a process to wait for a given semaphore, and SIGNAL informs the System that a semaphore is again available.

ATTACH associates a semaphore with an external interrupt. When that interrupt occurs, the semaphore is signalled. A process may synchronize with the interrupt by waiting on the semaphore.

The use of these intrinsics is demonstrated in examples below.

VIII.3 Mutual Exclusion

When concurrent processes must share resources, it may often be essential for only one process to access a particular resource at a given time. This is known as "mutual exclusion". It may be achieved by allowing the resource to be accessed only in "critical sections" of code to which the mutual exclusion criteria are applied.

Suppose, for example, that two processes must both display information on the console and request input from the operator, but only one process may be allowed to do so at a time. These two processes must therefore practice mutual exclusion with respect to the operator's console.

Critical sections may be implemented using Boolean semaphores by enclosing the critical section between WAIT(sem) and SIGNAL(sem). The semaphore should be initialized to 1.

Example:

```
Initialize: SEMINIT( bridge_empty, 1 );

Critical Section:
                Procedure CROSSBRIDGE;
                  begin
                    WAIT( bridge_empty );
                         .
                         .
                         {critical section code}
                         .
                         .
                    SIGNAL( bridge_empty );
                  end (* CROSSBRIDGE *);
```

In this example, processes (e.g., "trains") seeking to use the critical section (e.g., to cross a bridge that holds only one train at a time) will simply call CROSSBRIDGE, which takes care of mutual exclusion internally via the global semaphore bridge_empty.

VIII.4 Synchronization

When concurrent processes are cooperating, the programmer will frequently want one process to wait at a certain point in its execution until another process has caused some event to occur, such as filling a buffer. A counting semaphore may be used as an "eventname" in this case. In the following example, two distinct "events", the filling or emptying of a buffer, are used to synchronize two concurrent processes.

Example:

```
PROGRAM BLUFF;
  const    N = (* Number of available buffers *);
  var      buff_full, buff_avail : semaphore;

PROCESS FILL_BUFFER;
  begin
    repeat
      wait( buff_avail );
      .
      (* Select and fill a buffer *)
      .
      signal( buff_full )
    until false;
  end;

PROCESS SEND_BUFFER;
  begin
    repeat
      wait( buff_full );
      .
      (* Select and send a buffer *)
      .
      signal( buff_avail )
    until false;
  end;

begin (* BLUFF *)
  seminit( buff_full, 0 );
  seminit( buff_avail, N );
  start( FILL_BUFFER );
  start( SEND_BUFFER );
  .
  .
end.
```

VIII.5 Event Handling

Event handling allows Pascal programs to respond to low level events. These events include such things as hardware interrupts and low level I/O actions.

The print spooler, for example, is able to function concurrently with other p-System activity through this mechanism. Every time a character is entered at the keyboard, an event occurs (on those systems set up for print spooling). The software can take advantage of this knowledge and switch back and forth between an editing session, for example, and the print spooling process.

In order to understand event handling, it is necessary to understand Pascal processes. A process is a routine similar to a function or procedure. Processes may run concurrently with each other and with the main program. Once a process is STARTed, it may WAIT on a semaphore before continuing execution. While a process waits, other processes (including the main program) may resume execution. When the semaphore in question is SIGNALed, the waiting process may have the opportunity to resume execution.

An event is assigned a particular number. System events use the numbers 0..31, and user-defined events may use the numbers 32..63. To handle events other than event 19 (used for print spooling), you must augment the SBIOS with calls to the BIOS routine EVENT with the desired event number as parameter.

The Pascal programmer should use the ATTACH intrinsic to assign event numbers to semaphores. When an event occurs, the p-System SIGNALs the corresponding semaphore. If no semaphore is ATTACHed to the event number, no action is taken.

Therefore, the occurrence of a low level event may possibly cause a particular Pascal process to receive processor time.

The following example shows how events might be used. This program assumes that the SBIOS has been set up to cause event 32 whenever input is received from a special user device. It executes normally until the user device input is noticed, at which point it handles that input and returns to normal execution.

```
PROGRAM USER_EVENT;
CONST
  EVENT_NUM = 32;
VAR
  PID: PROCESSID;
  S: SEMAPHORE;
  FINISHED: BOOLEAN;
        .
        .
        .
PROCESS HANDLE_USER_INPUT;
BEGIN
  WHILE TRUE DO
    BEGIN
      WAIT(S);
      IF FINISHED THEN EXIT(HANDLE_USER_INPUT);
        .
        .
        .
    END
END;

BEGIN
  FINISHED:=FALSE;
  SEMINIT(S,0);
  ATTACH(S,EVENT_NUM);
  START(HANDLE_USER_INPUT,PID,300,200);
      .
      .
      .
  FINISHED:=TRUE;
  SIGNAL(S);
END.
```

In the preceding example, the semaphore S is ATTACHed to event number 32. Then process HANDLE_USER_INPUT is STARTed with a high priority of 200. HANDLE_USER_INPUT then WAITs on S while the rest of the program continues normal execution. Whenever the SBIOS notices input from the user device, event 32 is caused. This event SIGNALs S, and because HANDLE_USER_INPUT has a high priority, it immediately receives processor time to take care of the input.

Each time HANDLE_USER_INPUT returns to the beginning of the loop, it WAITs on S and the main program resumes. Finally, the boolean FINISHED is set to true by the main program and S is SIGNALed directly. This causes an exit from the loop within HANDLE_USER_INPUT, and the program terminates.

NOTE: An occurrence of ATTACH using a given event number will override any previous occurrence of ATTACH with that event number. This means that the first semaphore will no longer be ATTACHed to the event.

NOTE: To detach a previously attached semaphore from an event number, attach NIL to that event number. For example, Attach(NIL,EVENT_NUM); detaches S from EVENT_NUM in the preceding program example.

VIII.6 Other Features

As noted above, there is a predefined type PROCESSID; a value of type PROCESSID may be returned upon the invocation of a process. In the present implementation, processid's are not considered a user-oriented feature, but are used for Operating System work. Variables of type processid may be used in expressions in the same way as pointer variables. That is, only the operators <>, =, and := are legal.

All processes must be declared at the outer (global) block of a program. They may not be declared within a procedure or another process. Process initiation must occur in the principal task of a program. That is, a process may not be started from any of a program's subsidiary processes.

Users interested in using processes at a fairly low level, especially using them in conjunction with the System's facilities for memory management and Heap control, should refer to the Internal Architecture Guide for further details.

IX. UTILITIES

The UCSD System's utilities are various precompiled programs that may be run with the eX(ecute command. They supply some functions that are sufficiently useful to be included in the System, yet not used frequently enough to warrant their being included among the System commands.

The location of the various utilities on disks as shipped is given in the Introduction to the UCSD p-System and the Pocket Guide.

IX.1 Preparing Assembly Codefiles for Uses Outside of the System

The utility program COMPRESS inputs codefiles consisting of one or more linked assembly procedures, and produces object files suitable for applications outside of the UCSD p-System's runtime environment.

COMPRESS can produce either relocatable or absolute object files. Absolute codefiles are relocated to the base address specified by the user, and contain pure machine code. Relocatable codefiles include a simplified form of relocation information (a description of its format is in a following section). Both kinds of output files are stripped of all file information normally used by the System, and must be loaded into memory by the user (or a user program) in order to execute properly.

IX.1.1 Preparing Codefiles for Compression

The assembly procedure(s) must be assembled with the UCSD Adaptable Assembler, and linked with the Linker (seed Chapter VI, and Section VII.4). Codefiles containing anything other than one segment of linked assembly code will cause COMPRESS to abort. Routines to be compressed should not contain any of the following assembler directives:

.ORG .ABSOLUTE .PUBLIC .PRIVATE .CONST .INTERP

.ORG and .ABSOLUTE are intended for producing absolute codefiles directly from the assembler (see sections VI.2.3 and VI.2.8).

.PUBLIC, .PRIVATE, .CONST and .INTERP are expressly designed for communication between assembly procedures and a host compilation unit (whether Pascal or some other language). These have no intended uses outside of the System's runtime environment. Their inclusion in an assembly program generates relocation information in formats that will cause COMPRESS to abort.

IX.1.2 Running COMPRESS

The codefile name is COMPRESS.CODE. At the command level, eX(ecute COMPRESS. It will respond with the following prompt:

Assembly Code File Compress <release version>

Type '!' to escape

Do you wish to produce a relocatable object file? (Y/N)

If the characters 'Y' or 'y' are typed, the following prompt appears:

Base address of relocation (hex):

This is the starting address of the absolute codefile to be produced. It should be entered as a sequence of 1 to 4 hexadecimal digits followed by a <return>. The prompt will reappear if an invalid number is entered.

The following prompts always appear:

File to compress :

Enter the name of the file to be compressed. It is not necessary to type the '.CODE' suffix. If the file cannot be found, the prompt will reappear.

Output file (<ret> for same) :

Enter the name of the output file, which can be any legal filename (COMPRESS does not append a .CODE suffix). Typing a <return> here causes the output file to have the same name as the input file, thus eliminating the input file. If the file cannot be opened, COMPRESS will print an error message and abort.

In all the previous prompts, typing the character '!' causes COMPRESS to abort.

After receiving information from the prompts, COMPRESS reads the entire source file, compresses the procedures, and writes out the entire destination file. Large codefiles may cause COMPRESS to abort if the system does not have sufficient memory space.

While running, COMPRESS displays for each procedure the starting and ending addresses (in hex), and the length in bytes. After finishing, the total number of

bytes in the output file is displayed. If an absolute codefile was produced, the highest memory address to be occupied by the loaded codefile is displayed.

The output of COMPRESS is a file of pure code, which must be loaded and executed directly by user software.

IX.1.3 Action and Output Specification

COMPRESS removes the following information from input files:

The segment dictionary (block 0 of codefile).
Relocation list and procedure dictionary pointers.
Symbolic segment name and code sex word.
Embedded procedure DATASIZE and EXITIC words.
Procedure dictionary and number of procs word.
Standard relocation list.

Procedure code in the output file is contiguous, except for pad bytes which are emitted (when necessary) to preserve the word-alignment of all procedures. Codefiles contain integral numbers of blocks of data; space between the end of the actual code and the end of the codefile is zero-filled.

Relocatable object files have the following format:

The relocatable code is immediately followed by relocation information. The last word in the last block of the codefile contains the code-relative word offset of the relocation list header, e.g.:

<starting byte address of loaded code> + <word offset * 2>
= <byte address of relocation list header word>

The list header word contains the decimal value 256. The next-lower-addressed word contains the number of entries in the relocation list. This word is followed (from higher addresses to lower addresses) by the list of relocation entries.

Beneath the last relocation entry is a zero-filled word which marks the end of the relocation info. Each relocation entry is a word quantity containing a code-relative byte offset into the loaded code, e.g.:

<starting byte address of loaded code> + <byte offset>
= <byte address of word to be relocated>

Each byte address pointed to by a relocation entry is a word quantity which is relocated by adding the byte address of the front of the loaded code.

Important Note: If you are relocating your file towards the high end of the 16-bit address space, you must ensure that the relocated file will not wrap around into low memory (i.e., <relocation base address>+ <codefile size> must be less than or equal to FFFF(hex). COMPRESS performs no internal checking for this case.

IX.2 Patch

PATCH is a utility which allows hands-on viewing and altering of files. PATCH is meant for bit manipulation and other messy but sometimes useful tasks. It was written as a personal utility, but was quickly incorporated into the standard set of System tools.

PATCH is meant to be used interactively with a CRT. It uses the Screen Control Module (see the Internal Architecture Guide) to accomplish this, and is therefore terminal-independent (within the usual limitations -- again, refer to the Internal Architecture Guide).

There are two main facilities in PATCH: a mode for editing files on the byte level, and a mode for dumping files in various formats.

The byte-editing capability allows the user to edit not only textfiles, but also to do quick fixes to codefiles and create specialized test data.

The dump capability provides formatted dumps in various radices. It also allows dumps from main memory.

IX.2.1 EDIT Mode

When PATCH is first eX(ecuted, the user is in EDIT mode. DUMP is reached by typing 'D'. No information is lost in toggling back and forth between the two modes.

EDIT allows the user to open a file or device, read selected blocks (specified by relative block number) into an edit buffer, then either view that buffer, or modify it (with TYPE) and write the modified block back to the file. Buffers are displayed on the screen in desired format, and edited in a manner similar to the Screen Oriented Editor.

The individual commands of EDIT are explained in some detail below. When it is impossible to perform a command, PATCH responds with self-explanatory error messages.

The promptlines for EDIT are :

EDIT : D(ump, G(et, R(ead, S(ave, M(ix, T(ype, I(nfo, F(or, B(ack, ?

EDIT : V(iew, W(ipe, Q(uit, ?

D(ump - calls DUMP.

G(et - opens the file or device that one wishes to use, and reads block zero into the buffer.

R(ead - reads a specified block from the current file.

S(ave - writes the contents of the buffer out to the current block.

M(ixed - changes the display format for the current block. Typing 'M' toggles between the two formats:

Mixed - displays printable ASCII characters, and the hexadecimal equivalent of nonprintable characters;

Hex - displays the block in hexadecimal digits.

I(nformation - displays information about the current file.
This includes:
the filename,
the file length,
the number of the current block,
whether the file is open,
whether UNITREADs are allowed,
the device number (-1 if UNITIO is False),
the byte sex of the current machine.

F(orward - gets the next block in the file.

B(ackward - gets the preceding block in the file.

V(iew - displays the current block. (see M(ixed)

W(ipedisplay - clears the display of the block off the screen.

Q(uit - quits the PATCH program.

T(ype - goes into the typing mode, which allows the buffer to be edited. (described immediately below)

IX.2.2 TYPE Mode

TYPE, like the Screen Oriented Editor, allows the information on the screen to be modified by moving the cursor around and typing over existing information. If you make errors while using TYPE, do not S(ave the buffer while in EDIT mode, but R(ead the block over and try again.

The promptline for TYPE is:

TYPE : C(har, H(ex, F(ill, U(p, D(own, L(eft, R(ight, <vector arrows>, Q(uit

C(haracter - exchanges bytes in the buffer for ASCII characters as they are typed, starting from the cursor and continuing until an <etx> is typed. Only printable characters are accepted.

H(ex - exchanges bytes in the buffer for hex digits as they are typed, starting from the cursor and continuing until a 'Q' is typed. (Hex digits can be either upper or lower case.)

F(ill - fills a portion of the current block with the same byte pattern. Accepts either ASCII characters or hexadecimal digits for the pattern. When finished, the cursor will be positioned after the last byte filled.

The following commands move the cursor around within the block of data being displayed. The cursor is always at a particular byte. Rather than moving off the screen, the cursor wraps around from side to side and from top to bottom.

U(p - moves the cursor up one row

D(own - moves the cursor down one row

L(eft - moves the cursor left one column

R(ight - moves the cursor right one column

<vector arrows> - these are the vector arrows as used in the Screen Editor. They will do the same respective actions as U,D,L,R.

Q(uit - quits the TYPE mode and returns to the EDIT mode.

IX.2.3 DUMP Mode

Dumps can be generated in the following formats: decimal, hexadecimal, octal words, ASCII characters (if printable), decimal bytes (BCD), and octal bytes.

DUMP is also capable of flipping the bytes in a word before displaying it, or simultaneously displaying a line of words in both flipped and non-flipped form.

Input to DUMP can be from a diskfile specified by the user, or directly from main memory (this is primarily used to examine the Interpreter and/or the BIOS).

The width of the output can be controlled; a line may contain any number of machine words. 15 words fill an 132-character line, and 9 fill an 80-character line.

When the user enters DUMP, the screen shows a brief promptline - D(o it and Q(uit, and a lengthy menu of format specifications which are modifiable by typing the letter of the item and then entering the specification.

The Specifications:

A) : the input: a disk file or device.

B) : the number of the block from which dumping starts.
If (A) is a device, this number is not range-checked.

C) : the number of blocks to print out.
If this is too large, DUMP merely stops when there are no more blocks to output.

D) : Typing 'D' starts the dump.

E) : a toggle: if True, then reads from main memory,
if False, reads from the file in (A).

F) : an offset: the dump may start with a byte that is past byte zero. 0 <= (F) <= maxint.

G) : the number of bytes to print. 0 <= (G) <= maxint.

H) : the output file, opened as a textfile.

I) : the width of the output line, in machine words.
1 <= (I) <= 15.

The following six items have three associated Booleans that must be specified: USE, FLIP, and BOTH.

USE tells DUMP whether or not to use the format associated with that item.

FLIP tells DUMP whether or not to flip the bytes before displaying words in that format.

BOTH tells DUMP to simultaneously display both Flipped and non-Flipped versions of the line. If BOTH is True, the value of FLIP does not matter.

J) : display each word as a decimal integer

K) : display each word as hexadecimal digits in byte order

L) : display each word as an octal integer. This is the octal equivalent of (J).

M) : display each word as ASCII characters in byte order. Unprintable characters are displayed as hex digits.

N) : display each word as decimal bytes (BCD) in byte order.

O) : display each word as octal digits in byte order.

Q) : typing 'Q' returns to EDIT mode. DUMP remembers the current specifications.

S) : put a blank line after the non-Flipped version of a line.

T) : put blank lines between different formats of a line.

Both EDIT and DUMP modes remember all their pertinent information when the other mode is operating.

IX.2.4 A Note on Prompts

All user-supplied numbers used by PATCH are read as strings and then converted to integers. Only the first five characters of the string are considered. If there are any non-numeric characters in the string, the integer defaults to zero. If integer overflow occurs, the integer defaults to maxint. (Since integer overflow can only be detected by the presence of a negative number, integers in the range 65536 .. 98303 will come out modulo 32768.)

IX.3 The Decoder Utility

The Decoder utility, called DECODE.CODE, replaces the Disassembler and Libmap utilities used in previous versions. It provides access, in symbolic form, to all useful items contained in codefiles. Among the information available is the following:

1) Names, types, global data size, and other general information about all code segments in the file;

2) INTERFACE section text (if present) for all UNITs in the file;

3) Symbolic listing of any (or all) P-code procedures in any (or all) segments of the file;

4) Segment references and linker directives associated with code segments.

Decoder should be used whenever detailed knowledge of the internal contents of a codefile are desired (for instance, an implementor of a P-machine would decode test programs so that step-by-step execution of the object code could be done easily). The Internal Architecture Guide may be useful reading if detailed use of Decoder is planned.

If a program USES a UNIT, the UNIT will only be decoded if it is within the host file; Decoder will not search the disk for UNITs to decode. Assembly routines linked into a higher-level host will not be disassembled when the host is decoded.

When Decoder is eX(ecuted, the first prompt asks for the input codefile (the suffix .CODE is automatically appended if necessary). The next prompt asks for the name of a listing file to which Decoder's output may be written. This may be CONSOLE: (indicated by typing <return>), REMOTE:, PRINTER:, or a disk file. The following prompt is then displayed:

Segment Guide: A(ll), #(dct index), D(ictionary), Q(uit)

The D(ictionary option displays the code file's segment dictionary. A(ll disassembles all segments. A number of a dictionary index followed by <return> disassembles a given segment (if present), and Q(uit leaves the Decoder.

EXAMPLE:

Given the following Pascal program:

```
 1          0:d     1  {$L LIST1.TEXT}
 2          1:d     1  PROGRAM DEMO;
 3          1:d     1  VAR  1:INTEGER;
 4          1:d     2
 5          1:d     2  SEGMENT PROCEDURE ADD1;
 6    3     1:0     0  BEGIN
 7    3     1:1     0    1:=1+1;
 8    3     1:0     5  END;
 9    3     1:0     7
10    2     1:0     0  BEGIN
11    2     1:1     0    1:=50;
12    2     1:1     4    REPEAT
13    2     1:2     4      ADD1;
14    2     1:1     7    UNTIL 1=400;
15           :0    14  END.
```

... Decoder would prompt for input and output filenames. Then, if D(ictionary was typed, the following would be displayed:

```
INDEX NAME  START  SIZE  VERSION M_TYPE    SG#  SEG_TYPE   RL  FMY_NAME or
                                                              DSIZE  SGRF  HISG
  0   DEMO    2     20    IV.0  M_PSEUDO  2    PROG_SEG   R     1     10    4
  1   ADD1    1     14    IV.0  M_PSEUDO  3    PROC_SEG   R     DEMO
  2                                             NO_SEG
  3                                             NO_SEG
  4                                             NO_SEG
  5                                             NO_SEG
  6                                             NO_SEG
  7                                             NO_SEG
  8                                             NO_SEG
  9                                             NO_SEG
  10                                            NO_SEG
  11                                            NO_SEG
  12                                            NO_SEG
  13                                            NO_SEG
  14                                            NO_SEG
  15                                            NO_SEG
(C):
Sex: LEAST significant byte first
Segment Guide: A(ll, #(index of dictionary entry, Q(uit
```

... and Decoding A(ll of this program would produce the following disassembly:

```
DATA POOL:    SEGMENT=DEMO       PROCEDURE=   1   BLOCK=  2   BLOCK OFFSET=   0

   0:0013.   | 0000.  | 4544.ED| 4F4D.OM| 2020.   | 2020.  | 001E.  | 0000.  |

          block #  2   offset in block=  16
     OFFSET                                  HEX CODE
     0(000):    LDCB        50               8032
     2(002):    SRO          1               A501
     4(004):    CXG          3      1        940301
     7(007):    SLDO         1               30
     8(008):    LDCI       400               819001
    11(00B):    EQUI                         B0
    12(00C):    FJP          4               D4F6
    14(00E):    CXG          4      2        940402
    17(011):    RPU          0               9600

DATA POOL:    SEGMENT=ADDI       PROCEDURE=   1   BLOCK=  1   BLOCK OFFSET=   0

   0:000d.   | 0000.  | 4441.DA| 4944.ID| 2020.   | 2020.  | 0015.  | 0000.  |

          block #  1   offset in block =  16
     OFFSET                                  HEX CODE
     0(000):    SLDO         1               30
     1(001):    SLDO         1               30
     2(002):    ADI                          A2
     3(003):    SRO                          A501
     5(005):    RPU                          9600
```

Decoder's D(ictionary display is a pretty format of the codefile's segment dictionary. The following information is given:

INDEX is Decoder's name for each segment. Individual segments may be disassembled by typing their number followed by <return>; e.g., '0'<return> for this sample would cause only DEMO to be disassembled.

NAME contains the names of each segment.

START contains each segment's starting block (relative within the codefile).

SIZE is the length (in words) of each segment.

VERSION is the UCSD p-System version number of the segment.

M_TYPE is the machine type. Usually this is M_PSEUDO, indicating a P-code segment, but assembled segments will indicate a given machine.

SEG_TYPE can be: NO_SEG, PROG_SEG, UNIT_SEG, PROC_SEG, or SEPRT_SEG. NO_SEG is an empty segment "slot", PROG_SEG is a program segment, UNIT_SEG is a UNIT segment, PROC_SEG is a SEPARATE routine segment, and SEPRT_SEG is an assembled segment.

The RL columns indicate whether or not the segment is relocatable, and whether it needs to be linked. An 'R' indicates a relocatable segment. An 'L' indicates a segment that must be linked.

If the segment is declared within a program or unit, then the FMY_NAME column will contain its "family name", i.e., the name of the program or unit. Otherwise, the DSIZE SGRF HISG columns are displayed, and contain respectively the compilation module's data size, segment references, and maximum number of segments.

At the bottom of the screen, '(C):' is followed by whatever copyright notice the codefile may have.

The next line indicates the byte sex of the codefile.

The promptline is the last line on the screen.

The first line of the disassembled listing shows the segment name, procedure number, block number and block offset of the code for that segment and procedure.

The next line contains a variable number of words. Each word is displayed as a hexadecimal number, most-significant-byte-first, and is followed by a period. After the period is a character representation of the word (if printable). The first word is the PROCEDURE DICTIONARY POINTER, followed by the RELOCATION LIST POINTER, and then the eight byte segment name. After the segment name is a variable number of words. The next-to-last word is the segment's EXITIC, followed by its DATASIZE. If the codefile is for a least-significant-byte-first machine, the ordering of characters may be reversed. The information represented here is described more fully in the Internal Architecture Guide.

The disassembled code itself is displayed in blocks. The OFFSET column shows the offset in bytes from the front of the procedure (the count is in both decimal and hex). Then the P-code mnemonic is displayed, followed by the operands, if any,

and finally the HEX CODE for that particular instruction.

The OFFSET column corresponds to the fourth column in a compiled listing.

Jump operands are displayed as offsets relative to the start of the procedure, rather than IPC-relative (IPC = instruction program counter). This is to make the disassembly more readable. Thus, the operand shown is the offset of some line; in the example, the false jump (FJP) on line 12 shows 4, which means line 4 -- the CXG 3 1 instruction; the HEX CODE indicates that the offset is actually F6 (= -10) (which is IPC-relative).

If a single segment were to be disassembled (rather than using the A(ll) command), a line similar to the following would be displayed:

```
There is 1 procedure in segment DEMO.
Procedure Guide: A(ll), #(of procedure), L(inker info),
                 S(egment references), I(nterface text), Q(uit)
```

Selecting A(ll) will disassemble all of the procedures in the segment (in the example there is only one). Typing a number of a procedure followed by return will disassemble that procedure. L(inker information, S(egment references and I(nterface text may also be displayed if they are present.

For example, if the segment were a unit with interface text and '1' was typed, the following might be displayed:

```
Interface text for segment SOMEUNIT:

PROCEDURE A_PROC;
PROCEDURE ANOTHER_PROC(1:INTEGER);
FUNCTION A_FUNCTION:BOOLEAN;
IMPLEMENTATION
```

If the segment had references to other segments and 'S' was typed, the following might be displayed:

```
Segment references list for segment KERNEL:
14: ***                 5: SYSCMND
13: CONCURRE            4: DEBUGGER
12: PASCALIO            3: FILEOPS
11: HEAPOPS             2: SCREENOP
10: STRINGOP            0:
```

If the segment had linker information and 'L' was typed, the following might be displayed:

```
Linker information for segment SOMESEG:

SOMEPROC   EXTPROC srcproc=4 nparams=0 koolbit=false
```

IX.4 Duplicating Directories

It is a sometimes worthwhile precaution to keep a duplicate directory on a disk. In certain situations, this may help rescue directory information that is lost or garbled, and help restore a disk or the files on it to some desired state. The Z(ero command of the Filer will create a duplicate directory, and so will the MARKDUPDIR utility described below. Once a duplicate directory has been created, the Filer maintains it along with the primary directory. The COPYDUPDIR utility copies a duplicate directory into the primary directory location.

IX.4.1 COPYDUPDIR

This program copies the duplicate directory of a disk into the primary directory location. EX(ecute COPYDUPDIR. It asks for the drive in which the copy is to take place (e.g., 4). If the disk is not currently maintaining a duplicate directory, COPYDUPDIR tells you so. If the duplicate is found, then COPYDUPDIR asks if you are sure you want to destroy the directory in blocks 2-5. A 'Y' executes the copy; any other character aborts the program.

IX.4.2 MARKDUPDIR

MARKDUPDIR marks a disk that is not currently maintaining a duplicate directory so that it will.

The user must be sure that blocks 6..9 are free for use. If they are not, the user must re-arrange the files on the disk so as to make blocks 6..9 free. One can tell if they are available by doing an E(xtended listing in the Filer and checking to see where the first file starts. If the first file starts at block 6 or the first file starts at block 10 but there is a 4-block unused section at the top, then the disk has not been marked. If, however, the first file starts at block 10 and there are no unused blocks at the beginning of the directory, then the disk has already been marked, and a duplicate directory may already exist.

Example:

```
  SYSTEM.PASCAL          31    30-Aug-78     6      Codefile
  .
  .
  .
OR

  <unused>                4                   6
  SYSTEM.PASCAL          31    30-Aug-78     10     Codefile
  .
  .
  .
```

... both of the above cases indicate disks that have not been marked. Below is the directory of a properly marked disk:

```
  SYSTEM.PASCAL          31    30-Aug-78     10     Codefile
  .
  .
  .
```

To execute this program, e(Xecute MARKDUPDIR. It will ask which drive contains the disk to be marked (4 or 5). MARKDUPDIR checks to see if blocks 6-9 are free. If they seem to not be free, it asks if you are sure they are free? Typing 'Y' executes the mark, any other character aborts the program. Be sure that the space is free before marking it as a duplicate directory, otherwise file information will be irretrievably lost.

IX.5 XREF -- The Procedural Cross-referencer

IX.5.1 Introduction

The Procedural Cross-Referencer is a software tool to assist programmers in finding their way around Pascal program listings of non-trivial size. In keeping with a basic philosophy that software tools should have distinct and clear purposes, the function of the Referencer is to provide: a compact summary of the procedure nesting in a program; a list of the procedures and, for each, the procedures which call them; and a table of calls made by each procedure along with all non-local variable references. It thus provides information about the inter-procedural dependencies of a program.

XREF is an adaptation of a cross-referencer written by Arthur H.J. Sale of the University of Tasmania (it was published in "Pascal News", March, 1980). His program in turn was based on a program by A.J. Currie of the University of Southhampton.

IX.5.2 Referencer's Output

The Referencer produces five tables and an optional warnings file:

1. Lexical Structure Table: Static procedure nesting.

2. Call Structure Table: Procedures and the procedures that they call.

3. Procedure Call Table: Procedures and the procedures that call them.

4. Variable Reference Table: Each procedure and the variables it references.

5. Variable Call Table: Each variable and the procedures which reference or modify it.

6. Warnings File { if desired }: Indicates possible problems in the source program.

IX.5.2.1 Lexical Structure Table

The first table displays the lexical structure and the procedure headings. (The term procedure means procedure, function, process or program in this document unless otherwise stated.) As the input program is read, each heading is printed out with the line numbers of the lines in which it occurs. The text is indented so as to display the lexical nesting. (This indentation must sometimes be 'crunched' to fit on an output line.)

Referencer considers a procedure heading to be any text between the words 'Procedure', 'Function', 'Process', or 'Program', and the following semicolon. This isn't the Pascal definition, but is more useful in debugging programs. If these reserved words are embedded within comments, they are ignored.

IX.5.2.2 The Call Structure Table

The second table is produced after the program has been scanned completely, and is the result of examining the internal data. For each procedure listed in alphabetical order, the table holds:

- The line-number of the line on which its heading starts.

- Unless it was EXTERNAL or formal (and had no corresponding block), the line number of the BEGIN that starts it's statement-part.

- The characters 'ext' if the procedure has an external body (declared with a directive other than FORWARD), the characters 'fml' if it is a formal procedural or functional parameter, or 'eh?' if it is declared forward with no associated forward block or BEGIN. If a number appears, the procedure has been declared FORWARD and this is the line number of the line where the block of the procedure begins (i.e., the second part of the two-part declaration).

- A list of all user-declared procedures directly called by this procedure. (In other words, their call is contained in the statement-part.) The list is in order of occurrence in the text; a procedure is not listed more than once.

- A list of variables referenced by this procedure, and (if non-local) the procedure in which they were declared. If a variable is modified by an assignment, then it is printed with a leading '*'.

IX.5.2.3 The Procedure Call Table

This is a list of procedures, in alphabetical order, and, for each procedure, the procedures which call it.

IX.5.2.4 Variable Reference Table

This is a list of procedures, in alphabetical order, and for each procedure, the variables which that procedure examines or modifies in any way. If the variable is not local to the procedure in question, then the procedure in which it was declared is listed.

Variable references are shown in three forms:

<variable name> ::= a local variable

<procedure name> <variable name> ::= a variable defined in <procedure> which is used but not modified

<procedure name>*<variable name> ::= a variable defined in <procedure> which is modified.

IX.5.2.5 Variable Call Table

This table is of the form:

<procedure name> <variable name>: <procedure name> [<procedure name>]

The first procedure name is the procedure which owns the variable name, and the following procedure(s) either examine or modify that variable.

IX.5.2.6 Warnings File

A file of warning messages. There are three types of warnings:

'Symbol' may be undeclared line# xxxx
'Symbol' may not be initialized line# xxxx
Not standard, Nested comments line# xxxx

'Symbol' is an identifier, and xxxx is the number of the line on which it occurs.

Referencer only catches initializations done by replacement statements (':='), so

variables which are initialized by procedure calls (including READ, etc.) will be flagged as possibly uninitialized. There may be a surplus of such warning messages, depending on the program.

The 'Not standard, Nested comments' warning refers to the nesting of comments of different bracket types: (* like this { verstehen Sie? } *), which is accepted by the UCSD Pascal Compiler, but not the current ISO draft standard.

The warnings file may only be generated if the Variable Reference Table is also generated.

IX.5.3 Using Referencer

The Referencer has options that are user-defined at runtime. When the user eX(ecute's XREF, Referencer prompts for answers for the following questions:

- Length of the output line [40..132]:
  This is the length of the output line for the terminal/printer that you have available. Suggested output width is 80 characters.

- Input File:
  The name of the text file that contains the Pascal program to be Referenced. If the specified file cannot be successfully opened, the prompt is repeated until the user either types a valid input file name, or simply <return>. Typing an empty filename (<return>) exits Referencer.

- Is this a compiled listing? [y/n]:
  The program will read either .TEXT files containing Pascal source programs, or listing files generated by the Compiler. Using a compiled listing as input assures the user that the line numbers referenced will be synchronized with the line numbers generated by the Compiler.

- Do you want intrinsics listed? [y/n]:
  This allows identifiers such as 'WRITELN', 'PRED', 'GET', to be accepted as valid symbols. These are then cross- referenced as procedures listed outside the lexical nesting and therefore are not expected to have a 'BEGIN' associated with them. This includes the special UCSD intrinsics listed in the UCSD Pascal Reference Manual.

- Do you want initial procedure nestings? [y/n]:
  This causes the Lexical Structure Table to be generated. This

table shows the procedure headings and, for each procedure, the list of procedures which it calls.

- Do you want procedure called by trees? [y/n]:
  This option is offered only if the Lexical Structure Table is desired. A 'y' causes both the Call Structure Table and the Procedure Call Table to be generated. The Procedure Call Table lists each procedure, and all of the procedures which call it. (A warning is displayed if less than 10000 words of memory are available to generate these trees; no provision is made for possible stack overflow.)

- Do you want variables referenced? [y/n]:
  A 'y' causes the Variable Reference Table to be generated.

- Do you want variable called by trees? [y/n]:
  A 'y' causes the Variable Call Table to be generated.

- Do you wish warnings? [y/n]:
  'Y' causes the Warnings File to be generated. This option is offered only if the preceding selection was made.

- Please enter the name of the warning file:
  If warnings are selected, then you have the option of directing them to any file. If the file is a disk file, the name should have '.TEXT' appended to it.

- Output File:
  The name of the file to which you would like the output directed. If the file is a disk file, the name should have '.TEXT' appended to it.

The referencer expects to read a complete and syntactically correct Pascal program. Although results with syntactically incorrect programs are not guaranteed, Referencer is not sensitive to most flaws. It cares about procedure, function, and program headings, and about proper matching of BEGINs and CASEs with ENDs in the statement-parts.

Referencer does not try to format procedure and function headings; it leaves them as they were entered in the program, except for indentation alignment.

The tables are all as wide as the output line length (as specified by the user). Eighty characters is usually sufficient. For large programs, the first table (Lexical Structure Table) will be clearer with a larger print line.

IX.5.4 Limitations

As mentioned before, the behavior of Referencer when presented with incorrect Pascal programs is not guaranteed. However, it has been designed to be fairly robust, and there are few flaws that will cause it to fail. The most critical features, and therefore those likely to cause failure if not correct, are the general structure of procedure headings, and the correct matching of an END with each BEGIN or CASE in each statement-part (since this information is used to detect the end of a procedure).

If an error IS explicitly detected (and Referencer has very few explicit error checks and minimal error-recovery), a message is printed out that looks like this:

FATAL ERROR - No identifier after prog/proc/func - At Line No. ###

The line number displayed (###) is where the program ran into trouble; like all diagnoses this does not guarantee that the correct parentage is ascribed to the error. Processing continues for a while despite the fatal error, but only the Lexical Structure Table is produced.

Referencer is believed to accept standard Pascal programs, UCSD Pascal Programs, and UCSD Units, and process each correctly.

IX.6 The Symbolic Debugger

The symbolic debugger is a tool for debugging compiled programs, which can be called from the main system prompt line or can be called during the execution of a program (when a breakpoint is encountered). Using the symbolic debugger, memory may be displayed and altered, P-code may be single-stepped, and markstack chains may be displayed and traversed.

There are no promptlines explaining the debugger commands because such prompts would detract from the information displayed by the debugger itself. However, when a command is entered, the system displays several short prompts that may ask for information.

Many of the debugger commands require two characters (such as "L P" for L(ist P(code, or "L R" for L(ist R(egister). To exit the program after typing the first character, press <space> to recall the main mode of the debugger.

To use the debugger effectively, a user must be familiar with the UCSD P-machine architecture and must understand the P-code operators, stack usage, variable and parameter allocation, etc. These topics are discussed in the Internal Architecture Guide.

A compiled listing of the program is a helpful debugging tool. It helps the user determine P-code offsets and similar information and should be current.

The debugger is a low-level tool, and as such, must be used with caution. If the debugger is used incorrectly, the p-System can fail.

IX.6.1 Entering and Exiting the Debugger

Press D(ebugger to enter the debugger from the main system prompt line. If the debugger is entered in a fresh state, the system displays the following prompts.

```
DEBUG [version #]
(
```

A fresh state means that the debugger was not previously active, and no breakpoints are currently enabled. If the debugger is entered in a non-fresh state,

only the left parenthesis "(" appears.

Exit the debugger by pressing Q(uit, R(esume, or S(tep. The Q(uit option disables the debugger. If the debugger is re-invoked, it is in a fresh state. The R(esume option does not disable the debugger and execution continues from where it left off. The debugger is still active; and if it is re-invoked, it is in a non-fresh state. The S(tep option executes a single P-code and automatically re-invokes the debugger in a non-fresh state.

If a program is running under the debugger's R(esume command, it may force a return to the debugger by calling the HALT intrinsic. In fact, any run-time error causes a return to the debugger if the debugger is active while the program is running.

The debugger may be memlocked or memswapped (see the descriptions of those intrinsics) by using the M(emory command at the outer level. "ML" memlocks and "MS" memswaps the debugger.

IX.6.2 Using Breakpoints

To enter the debugger while a program is running, but not alter the program's code, use the debugger to set breakpoints. Press B(reakpoint and then use either the S(et, R(emove, or L(ist command. To set a breakpoint, press S(et after pressing B(reakpoint. There are, at most, five breakpoints numbered 0 through 4. The system displays four prompts asking for information. The first prompt is--

```
Set Break #?
```

Enter a digit in the range 0..4 and press <space>. The next prompt is--

```
Segname?
```

Enter the name of the desired segment and press <space>. The next prompt is--

```
Procname or  #?
```

Enter the number of the desired procedure and press <space>. The final prompt is--

```
Offset #?
```

Enter the desired offset within the procedure and press <space>. The system sets a breakpoint, and if that segment, procedure, and offset are encountered during execution resumption, the debugger is automatically re-invoked.

Use a compiled listing of the program to determine the location of the breakpoint. If no compiled listing is available, use the text file viewing facility.

To set a breakpoint that differs only slightly from the one most recently set, press <space> for the break number or segment. The system uses the previous breakpoint's information. For example, to break in the same segment and procedure but with a different offset, enter a space for everything except the offset.

To remove a breakpoint, press B(reakpoint, then press R(emove. The prompt,

Remove break #?

appears. To remove a breakpoint, enter its number followed by a <space>.

To list the current breakpoints, press B(reakpoint and then press L(ist.

IX.6.3 Viewing and Altering Variables

The V(ar command allows the system to display data segment memory. It is another two-character command that must be followed by G(lobal, L(ocal, I(ntermediate, E(xtended, or P(rocedure. If G(lobal or L(ocal is selected, the system displays the following prompt.

Offset #?

Enter the desired offset into the data segment.

If I(ntermediate is selected, the system displays the following prompt.

Delta Lex Level?

Enter the appropriate delta lex level for the desired intermediate variable.
If E(xtended is selected, the system displays the following prompt.

Seg #? Offset #?

Enter the appropriate segment number and offset number for the desired extended variable.

If P(rocedure is selected, the system may display an offset within a specified procedure. The following prompts are displayed in sequence.

Segment name? Procname or #? Varname or Offset #?

When any of these options are used, the system displays a prompt similar to the following line.

```
( 1) S=INIT P#1 VO#1 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa
```

This example is a portion of the local activation record for segment INIT, procedure 1, variable offset 1, at absolute hex location 2C1A. Following this, eight bytes are displayed, first in HEX and then in ASCII (a dash "-" indicates that the character is not a printable ASCII character).

To view surrounding portions of memory press V(ar. After a line has been displayed by the V(ar command, a "+" or "-" may be entered. This displays the succeeding or preceding eight bytes of memory.

The eight bytes that are currently displayed may be altered. If a "/" is typed, then the line may be altered in hex mode. If a "\" is typed, then the line may be altered in ASCII mode. When altering in hex mode, any characters that are to be left unchanged may be skipped by typing <space>. In the ASCII mode, any characters to be left unchanged may be skipped by typing <return>.

It is possible to change the frame of reference from which the global, local, and intermediate variables are viewed. This can be done by using the C(hain command. After "C" is typed the U(p, D(own and L(ist options are available. If "L" is typed, all of the currently existing mark stack control words are displayed, with the most recently created one first. An entry in the list resembles the following line.

```
(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA
msenv=FEE8
```

This corresponds to a mark stack control word with the indicated static link (msstat), dynamic link (msdyn), interpreter program counter (msipc), and erec pointer (msenv). The indicated segment (HEAPOPS), procedure (#3), and offset (#23) are the return point for the procedure call which created the MSCW.

If the U(p or D(own options are used, the frame of reference moves up or down one link and the frame of reference for variable listings (using the "V" command) changes accordingly.

IX.6.4 Viewing Text Files from the Debugger

To view a text file from the debugger, press F(ile. The system displays the following prompt:

```
Filename?  First line #?  Last line #?
```

Enter the name of the text file to be viewed followed by <space>. The .TEXT portion of the file name is optional. Then enter the first and last line numbers that delimit the portion of text that the user wishes to view. This command lists as many lines as possible in the window from "first line" to "last line" of the indicated file.

The F(ile command is useful for debugging (especially using symbolic debugging) when a hard copy of the relevant compiled listing is not available. Using this command, the user can view source files on disk and disk files containing compiled listings without leaving the debugger.

IX.6.5 Displaying Useful Information

Whenever control is returned to the debugger (e.g. after a single step operation, or when a breakpoint is encountered), it displays various information if it is desired. This information may include P-machine registers, the current P-code operator, the information in the current markstack, or any specified memory location. In order to select what will be displayed, the E(nable mode should be used. After typing 'E', the following options are available at the command level, R(egister, P(code, M(arkstack, A(ddress, and E(very (all of the preceding). Any or all of these options may be enabled at the same time.

If R(egister is enabled, a line is displayed after each single step. The following line is an example of that display.

```
(rg) mp=F082 sp=F09C erec=FEE8 seg=9782 ipc=01C3 tib=0493 rdyq=2EBC
```

If P(code is enabled, a line such as the following is displayed after each step:

```
(cd) S=HEAPOPS P#3 O#23  LLA 1
```

If M(arkstack is enabled, a line such as the following is displayed after each step:

```
(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA
msenv=FEE8
```

If A(ddress is enabled, the system generates a display like the following line.

```
(a ) S=HEAPOPS P#3 O#23 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa
```

To initialize this address to a given value, use A(ddress mode at the outer level. Press A(dress and the system displays the following prompt.

```
Address ?
```

Enter the absolute address in hex. The system displays eight bytes starting at that address. Also, that address is now displayed if the E(nable A(ddress option is on.

Enabling E(very causes all of the above options to be enabled.

The D(isable mode disables any of the options just described. The L(ist mode lists any of the above options.

IX.6.6 Disassembling P-code

At the debugger's outer level, there is a P-code option that displays the P-code mnemonics for selected portions of code. This option asks for:

```
Segname?
Procname or #?
Start Offset #? and End Offset #?
```

The indicated portion of code is then disassembled. This may be useful during single-step mode if you wish to look ahead in the P-code stream. This mode may be exited before it reaches the ending offset by typing <break>; control returns to the debugger.

IX.6.7 Example of Debugger Usage

Suppose the following program is to be debugged:

```
Pascal Compiler IV.0

1 0 0:d 1  {$L LIST.TEXT}
2 2 1:d 1  PROGRAM NOT_DEBUGGED;
3 2 1:d 1  VAR I,J,K:INTEGER;
4 2 1:d 4      B1,B2:BOOLEAN;
5 2 1:0 0  BEGIN
6 2 1:1 0    I:=1;
7 2 1:1 3    J:=1;
8 2 1:1 6    IF K <> 1 THEN WRITELN ('Whats wrong?');
9 2  :0 0  END.

End of Compilation.
```

First we enter the debugger and set a breakpoint at the beginning of the IF statement:

```
(BS) Set break #? 0 Segname? NOTDEBUG Procname or #? 1 Offset #? 6
(EP)
(R)
```

After setting the breakpoint we enable P-code (EP) and resume (R). Now we execute the program above, and when it reaches offset 6, the debugger is entered. We single-step twice:

```
Hit break #0 at S=NOTDEBUG P#1 O#6
 (cd)  S=NOTDEBUG  P#1 O#6  SLDO1
 (cd)  S=NOTDEBUG  P#1 O#7  SLDC1
 (cd)  S=NOTDEBUG  P#1 O#8  NEQUI
```

We see that our first single-step did a short load global 1. (Note: This put K on the stack. K is NOT global 3; I is global 3, J is global 2, and K is global 1. Every string of variables (such as 'I, J, K' in a declaration) is allocated in reverse order. Boolean B1, which follows, is at offset 5, and B2 is at offset 4. Parameters, on the other hand, ARE allocated in the order in which they appear.) The second single-step did a short load constant 1 onto the stack. Now we are about to do an integer comparison (<>). But this is where our error shows up, so we decide to look at what is on the Stack before doing this comparison:

```
(LR)
(rg)  mp=EB62  sp=EB82  erec= ...
(A )  Address? EB82
(a )              EB82: 01 00 C5 14 ...
```

We list the registers and then look at the memory address to which register sp points. We discover a 1 on top of the stack (01 00: this is a least-significant-byte-first machine) followed by a word of what appears to be garbage. This leads us to suspect that K was not initialized. Looking over the listing, we quickly realize that this is the case.

IX.6.8 Symbolic Debugging

The symbolic debugging feature allows specification of variables by name, rather than P-code offset. Also, breakpoints and portions of code to be disassembled may be indicated by procedure name and line number, rather than procedure number and P-code offset.

Having a current compiled listing of the code in question is still essential for serious debugging efforts.

To use symbolic debugging, it is necessary that the code being debugged is compiled with the $D+ option. The $D+ option (Pascal only), which defaults to $D-, instructs the compiler to output symbolic debugger information for those portions of a program that are compiled with $D+ turned on. Once a program is debugged, it should be recompiled without symbolic debugger information, because this information increases the size of the code file.

Using symbolic debugging, breakpoints may be specified by procedure name and line number for all statements covered by the $D+ option. The B(reakpoint command will request:

"Procname or #?"

Enter the first eight characters of the procedure name. The next line displayed will be--

"First#__ Last#__ Line#?"

The underlines will actually be values that define the range of line numbers available to you within the specified procedure. (These line numbers appear on compiled listings.) You should then enter the desired line number for your breakpoint.

Variables within a given routine may be specified by name (rather than data segment offset number) if at least one statement within that routine is compiled by $D+. The V(ar command allows specification of G(lobal, L(ocal, I(ntermediate, or P(rocedure variables in this manner. E(xtended variables are not allowed to be specified symbolically. The V(ar command will prompt:

"Varname or Offset #?"

You may enter the first eight characters of the declared identifier. A line similar to the following will then appear:

(1) S=INIT P=FILLTABL V=TABLE1 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa

The segment is INIT, the procedure is FILL_TABLES, and the variable is TABLE1.

Similarly, the code to be disassembled by the P-code command can be specified symbolically for all portions of code covered by the $D+ option. This command will request:

Procname or #?

Enter the first eight characters of the procedure name. You will then be prompted as follows:

```
First #__ Last #__ Start Line#?  End Line#?
```

The underlines will actually be the boundaries that are available to you. You should enter the desired starting and ending line numbers. The specified code will then be disassembled.

IX.6.9 Symbolic Debugging Example

To use symbolic debugging, some part of a Pascal compilation unit must be compiled with the {$D+} compile-time directive. After this code has been generated, it is possible to reference variables and procedures by name rather than offset. The following example is a small Pascal program that has been compiled with the 'D' option.

```
=====================================================

Pascal Compiler IV.1 c5s-4      3/ 4/82      Page   1

=====================================================

    1   0    0:d    1      {$D+}
    2   2    1:d    1       program example;
    3   2    1:d    1       var a,b,c:integer;
    4   2    1:d    4
    5   2    1:d    4         procedure set_c_if_d;
    6   2    2:d    1         var d:boolean;
    7   2    2:0    0         begin
    8   2    2:1    0           d:=a>b;
    9   2    2:1    5           if d then
   10   2    2:2    8             c:=a*b;
   11   2    1:0    0         end;
   12   2    1:0    0
   13   2    1:0    0       begin
   14   2    1:1    0         a:=0;
   15   2    1:1    3         b:=5;
   16   2    1:1    6         set_c_if_d;
   17   2     :0    0       end.

End of Compilation.
```

The following listing is an example of a debug session.

```
Debug [x15]
(BS) Segname=EXAMPLE Procname or # = SETCIFD
       symbolic seg not in mem Line#? 8
(R )

Hit break#0 at S=EXAMPLE P=SETCIFD L#8
(BS) Segname=EXAMPLE Procname or # = SETCIFD
       First#8 Last#10 Line#? 9
(R )

Hit break#1 at S=EXAMPLE P=SETCIFD L#9
(VL) Varname or offset#? D
(1 ) S=EXAMPLE P=SETCIFD V=D  E7B2 : 0000 9448 BEE7 190C-H-
(Q )
```

The first time the debugger is entered, the program example is not in memory and hence the symbolic segment is not in memory. However, a breakpoint can still be set symbolically providing the user knows on which line number to stop. For the second breakpoint, the symbolic segment is in memory; because of this, its first and last line numbers are given.

Notice the variable 'D' was accessed symbolically, and its contents are displayed.

If the user tries to access symbolically when the actual code segment is in memory and its symbolic segment counterpart is not present, the system will display the error message 'symbolic seg not in mem'. Use the 'Z' command in the symbolic debugger to find out if symbolic information is available for a particular segment.

The 'Z' command lists all of the principal segments with their environment list. For example, the following example is a partial list of the principal segments (EDITOR and EXAMPLE) and their environments. The lowercase name 'example' is the symbolic segment for 'EXAMPLE'. The existence of 'example' indicates that symbolic debugging information is available for at least one procedure in 'EXAMPLE'.

```
(Z ) the sib is EDITOR
  1 KERNEL
  2 EDITOR
  3 INITIALI
  4 PASCALIO
  5 EXTRAIO
  6 GOTOXY
  7 STRINGOP
  8 EXTRAHEAP
  9 FILEOPS
  10 OUT
  11 COPYFILE
  12 ENVIRONM
  13 PUTSYNTA
  14 EDITCOR

the sib is EXAMPLE
  1 KERNEL
  2 EXAMPLE
  3 example
```

IX.6.10 Interacting with the Performance Monitor

The "I" command will call the PM_Interactive procedure within the operating system if the performance monitor is enabled. The user may, in this way, gain access to fault handling data, and information concerning what programs have started and completed execution.

IX.6.11 Summary of the Commands

A(ddress	Displays a given address
B(reak point	Segment, procedure and offset must be specified
S(et	Allows a break point (0 through 4) to be set
R(emove	Allows a break point to be removed
L(ist	Lists current break points
C(hain	Changes frame of reference for V(ariable command
U(p	Chains up mark stack links
D(own	Chains down mark stack links
L(ist	Lists current mark stacks
F(ile	Allows viewing of text files
E(nable	Enables the following to be displayed
D(isable	Disables the following from being displayed
L(ist	Lists the following
R(egister	The registers: mp, sp, erec, seg, ipc, tib, rdyq
P(code	Current P-code mnemonic
M(arkstack	Mark stack display
A(ddress	A given address
E(very	All of the above
I(nteractive	Interacts with the performace monitor
M(emory	
L(ock	Memlocks the debugger
S(wap	Memswaps the debugger
P(code	Dissassembles a given procedure
Q(uit	Quits the debugger, 'fresh' state if re-entered
R(esume	Exits debugger, debugger remains active, 'non-fresh'
S(tep	Single steps P-code and returns to debugger

V(ariable

G(lobal	Displays global memory
L(ocal	Displays local memory
I(nter	Displays intermediate memory
P(roc	Displays data segement of given procedure
E(xtended	Displays variables in another segment
Z(	Displays segment lists.

IX.7 The RECOVER Utility

RECOVER is a utility which attempts to re-create the directory of a disk whose directory has accidentally been destroyed.

RECOVER displays several yes/no prompts. These must be answered with upper-case letters: lower-case letters are ignored.

Following is a list of RECOVER's prompts, with a description of appropriate responses:

> RECOVER - Version IV.0.9
> ENTER TODAY'S DATE MM-DD-YY
>
> ... the user should enter a valid date, followed by <return>. Entering an incorrect date may cause RECOVER to abort with a value range error. Once a hyphen has been typed, it may not be backed over -- previous portions of the date may not be changed. The date that is entered is assigned to any files that RECOVER finds, which were not in the directory.
>
> USER'S DISK IN DRIVE:
>
> ... the user should type the number of the drive which contains the disk to be RECOVERed (i.e., a number in [4, 5, 9..22] followed by <return>).
>
> USER'S VOLUME ID:
>
> ... the user should type a volume name, which is recorded on the disk. The name should be in upper-case letters. Lower-case letters are accepted, but then the volume name is recorded with lower-case letters, which contradicts System standards.
>
> HOW MANY BLOCKS ON DISK?
>
> ... this prompt is only displayed if the number of blocks recorded in the (damaged) directory is not a valid number. The response depends on the disk drives used: respond as you would to the Filer's Z(ero command.

At this point, RECOVER reads each entry in the disk's directory, and checks it for validity. Entries with errors are removed. Entries that are valid are saved, and

RECOVER displays: 'ENTRY.NAME found' (or something similar).

When all the directory entries have been checked, and either saved or discarded, RECOVER prompts:

Are there still IMPORTANT files missing (Y/N)?

... responding 'N' causes RECOVER to prompt:

GO AHEAD AND UPDATE DIRECTORY (Y/N)?

... a 'N' exits RECOVER without doing anything.
... a 'Y' causes the reconstructed directory to be saved.

RECOVER displays:

WRITE OK

... and then terminates.

On the other hand, a 'Y' response to the 'Are there still IMPORTANT files missing?' prompt causes RECOVER to search those areas of the disk still not accounted for by the (partially) reconstructed directory. Textfiles and codefiles are detected, and appropriate directory entries created for them. If RECOVER cannot determine the original name of a textfile it has found, it creates a directory entry for DUMMY##.TEXT or DUMMY##.CODE (where the ## are two unique digits). If a codefile has a PROGRAM name, it is given that name; if this would create a duplicate entry in the directory, digits are used (for example, RECOVER restores first SEARCH.CODE, and then SEARCH00.CODE).

Data files cannot be detected by RECOVER, since their format is not System-defined. To recover data files, a user must resort to the PATCH utility (described in this chapter).

If RECOVER restores a textfile with an odd number of blocks, this probably means that the end of the textfile was lost: the user should use the Editor to make sure.

RECOVERed codefiles should be L(inked again, if that was originally necessary.

When RECOVER has finished its pass over the entire disk, it prompts:

GO AHEAD AND UPDATE DIRECTORY (Y/N)?

... and so forth, as described above.

IX.8 Turtlegraphics

Turtlegraphics is a package of routines for creating and manipulating images on a graphic display. These routines can be used to control the background of the screen, draw figures, alter old figures, and display figures using viewports and scaling. It also contains routines that allow the user to save figures in disk files and retrieve them.

The simplest turtlegraphic routines are intentionally very easy to learn and use. Once the user is familiar with these, more complicated features (such as scaling and pixel addressing) should present no problem.

A pixel is a single picture element or point on the display.

Turtlegraphics allows the user to create a number of figures, or drawing areas. One such figure is the display screen itself, and other figures can be saved in memory. Each figure has a turtle of its own. The size of a figure may be set by the user (it does not need to be the same size as the actual display).

The actual display is addressed in terms of a display scale, which may be set by the programmer. This allows the user's own coordinates to be mapped into pixels on the display. All other figures are scaled by the global display scale.

The programmer may also define a viewport, or window on the display. This limits all graphic activity to within that port.

The unit Turtlegraphics is already installed in *SYSTEM.LIBRARY, and a program may use its routines by including the following declaration.

USES turtlegraphics; (or an equivalent declaration in FORTRAN).

IX.8.1 Using Turtlegraphics

Each subsection below is divided into two parts. The first part is an overview of the topic at hand, and the second part consists of descriptions of the relevant turtlegraphics routines.

For quick reference this chapter contains a listing of the interface part of the turtlegraphics unit and a sample program that illustrates a number of the turtlegraphics routines.

IX.8.1.1 The Turtle

The turtle is an imaginary creature in the display screen that will draw lines as the user moves it around the display. The turtle can move in a straight line (Move), move to a particular point on the display (Moveto), turn relative to the current direction (Turn), and turn to a particular direction (Turnto).

Thus, the turtle draws straight lines in some given direction. The color of the lines it draws can be specified (Pen_color), and so can the nature of the line drawn (Pen_mode).

Wherever the turtle is located, its position and direction can be ascertained by three functions: Turtle_x, Turtle_y, and Turtle_angle.

NOTE: The turtle may be moved anywhere; it is not limited by the size of the figure or the size of the display. But, only movements within the figure will be visible.

To use the turtle in a figure other than the actual display, the programmer may call Activate_Turtle.

The following paragraphs describe the routines that control the turtle.

Procedure Move (distance: real);

Moves the active turtle the specified distance along its current direction. The turtle leaves a tracing of its path (unless the drawing mode is 'nop'). The distance is specified in the units of the current display scale (see below). The movement will be visible unless the current turtle is in a figure that is not currently on the display.

Procedure Moveto (x,y: real);

Moves the active turtle in a straight line from its current position to the specified location. The turtle leaves a tracing of its path (unless the drawing mode is 'nop'). The x,y coordinates are specified in the units of the current display scale.

Procedure Turn (rotation: real);

Turns the active turtle by the amount specified (in degrees). A positive angle turns the turtle counterclockwise, and a negative angle turns it clockwise.

Procedure Turnto (heading: real);

Sets the direction (the heading) of the active turtle to a specified angle. The angle is given in degrees; zero (0) degrees faces the right side of the screen, and ninety (90) degrees faces the top of the screen.

Procedure Pen_color (shade: integer);

Selects the color with which the active turtle traces its movements (unless the pen mode is 'nop'). This color remains the same until Pen_color is called again.

The HP-86/87 has two colors:

0 = Black
1 = White

Turtlegraphics uses a numeric designation for color instead of a symbolic designation like the word blue or red to maintain the p-System language and hardware compatibility. For example, while Pascal would allow the use of symbolic color designations, FORTRAN would not.

Procedure Pen_mode (mode: integer);

Sets the active turtle's drawing mode. This mode does not change until Pen_mode is called again.

These are the possible modes:

0 = Nop - does not alter the figure.

1 = Substitute - writes the current pen color.

2 = Overwrite - writes the current pen color.

3 = Underwrite - writes the current pen color. When the pen crosses a pixel that is not of the background color, that figure is not overwritten.

4 = Complement - the pen complements the color of each pixel that it crosses. (The complement of a color is its opposite: the complement of the complement of a color is the original color.)

Values greater than 4 are treated as Nop.

These descriptions apply to movements of the turtle. They have a more complex meaning when a figure is copied onto a figure that is already displayed.

Function Turtle_x : real;

Returns a real value that is the x-coordinate of the active turtle, in units of the current Display_scale.

Function Turtle_y : real;

Returns a real value that is the y-coordinate of the active turtle, in units of the current Display_scale.

Function Turtle_angle : real;

Returns a real value that is the direction (in degrees) of the active turtle.

Procedure Activate_Turtle (screen: integer);

Specifies to which figure subsequent turtlegraphics commands are directed. Each invocation of this procedure puts the previously active turtle to sleep and awakens the turtle in the designated figure. When turtlegraphics is initialized, the turtle in the actual display is awake. The initial position of the turtle is (0,0) or the bottom left corner of the screen, ready to move right.

IX.8.1.2 The Display

We refer to the initial background of the display as the wild card color. The wild card color (color 0) default is black. The background color of a turtlegraphics figure may be changed by the programmer with a call to Background. This "soft" background applies when drawing mode is used.

A figure can be filled with a single color (not necessarily the background color) by calling Fillscreen.

NOTE: Unless the user interrupts a program with the Break keystroke ([CTRL][P]), turtlegraphics will exit with a call to Alpha_All_Mode before returning to the System command level. Refer to the following paragraphs for a description of the four HP-86/87 display modes.

Procedure Fillscreen (screen: integer;
shade: integer);

Fills the specified figure ("screen") with the specified color ("shade"). If screen = 0, which indicates the actual display screen, then only the current viewport is shaded. For user-created figures, the entire figure is shaded.

Procedure Background (screen: integer;
shade: integer);

Specifies the backgound color for a figure. The initial background color of all figures is the wild card color.

There are four procedures in unit Usergraphics to utilize the four screen modes available on the HP 86/87: Alpha, Graph, Alphall, and Graphall. They allow the Pascal program to switch between graphics and alpha display and to dimension the graphics screen. Usergraphics, which exists in SYSTEM.LIBRARY, must be included in the Uses statement with unit Turtlegraphics.

Example:

```
Program TGR;
Uses Usergraphics, Turtlegraphics;
Begin
  ...
End.
```

Note, however, that the Usergraphics unit need not be specified if you have no intention of switching among the four HP-86/87 display modes during program execution.

When a program using Turtlegraphics is executed, the display will automatically be changed to GRAPH mode with a screen dimensions of 0..399 in the X direction and 0..239 in the Y direction. Exiting the program returns the display to ALPHALL mode and clears the graphics screen.

The following display mode changes will clear both the alpha and graphics screens:

1) From Alphall to any other mode.
2) From any other mode to Alphall.
3) From Graphall to any other mode.
4) From any other mode to Graphall.

Switching between Alpha Mode and Graph Mode will not affect the contents of either screen.

When switching to a new graphics mode, it is necessary to do a display scale.

EXAMPLES:

```
Graph_All_Mode;
Display_Scale(0,0,543,239);
      .
      .
Graph_Mode;
Display_Scale(0,0,399,239);
```

Procedure Graph_Mode;

The screen dimensions are 0..399 (X dimension) by 0..239 (Y dimension) pixels. Switching from Graph_All_Mode or Alpha_All_Mode will clear the graphics screen.

Procedure Graph_All_Mode;

The screen dimensions are 0..543 (X direction) by 0..239 (Y direction) pixels. Switching from Graph_Mode, Alpha_Mode, or Alpha_All_Mode will clear the graphics screen.

Procedure Alpha_Mode;

The screen dimensions are 80 columns by 24 rows of characters. Switching from Graph_All_Mode or Alpha_All_Mode will clear the alpha screen.

Procedure Alpha_All_Mode;

The screen dimensions are 80 columns by 24 rows of characters. Switching from Graph_Mode, Graph_All_Mode, or Alpha_Mode will clear the alpha screen.

For more information on display modes, refer to Introduction to Graphics, HP-86/87 Operating and BASIC Programming Manual.

Note: Echoed characters in Graph_Mode or Graph_All_Mode will cause stray pixels to be turned off or on. In Pascal, characters are echoed as a result of READ and READLN statements from standard INPUT and as a result of WRITE and WRITELN statements. To avoid side effects, use KEYBOARD--the non-echoing equivalent of INPUT--for READ and READLN statements, and use the WCHAR and WSTRING Turtlegraphics procedures instead of WRITE and WRITELN statements. For example, READLN(KEYBOARD,NameString) will not echo the input.

IX.8.1.3 Labels

It is possible to draw legends, labels, and so forth on the display while using the turtlegraphics unit. This is done by calling either WChar or WString. The character or string appears at the location of the currently active turtle. The text is displayed in the type font defined by the file *SYSTEM.FONT.

Procedure WChar (c: char; copymode, shade: integer);

Writes a single character at the position of the currently active turtle, using the indicated pen mode and color. The character is always displayed horizontally, regardless of the active turtle's direction.

Procedure WString (s: string; copymode, shade: integer);

Writes a string starting at the position of the currently active turtle, using the indicated pen mode and color. The string is always displayed horizontally, regardless of the active turtle's direction.

IX.8.1.4 Scaling

When a programmer wishes to display data without altering the input data itself, it is possible to set scaling factors that translate data into locations on the display. This is done with Display_scale. The display scale applies globally to all figures.

Because of the shape of the actual display, data for particular shapes (especially curved figures) might become distorted when using a "straight" display scale. In this case, the function Aspect_ratio can be used to preserve the "squareness" of the figure.

**Procedure Display_scale
(min_x,min_y,max_x,max_y: real);**

Defines the range of input coordinate positions that are to be visible on the display. Turtlegraphics maps the user's coordinates into pixel locations according to the scale specified in Display_scale.

This procedure sets the viewport to encompass the whole display. The display bounds apply to input data. For the actual display, these bounds can be any values the user requires, but for user-created figures (0,0) is the lower left-hand corner.

The default display scale is:

```
min_x = 0, max_x = 399
min_y = 0, max_y = 239
```

As an example, if a user wishes to graph a financial chart from the years 1970 to 1980 along the x axis, and from 500,000 to 500,000,000 along the y axis, the following call could be used.

```
Display_scale(1970, 5.0E5, 1980, 5.0E8)
```

After this, calls to turtle operations could be done using meaningful numbers rather than quantities of pixels.

Note that you must include a call to Display_scale when changing to Graph_All mode for the first time or when changing back to Graph mode.

Function Aspect_ratio : real;

Returns a real number that is the width/height ratio of the CRT. This can be used to compute parameters for Display_Scale that provide square aspect ratios.

If an application is designed to show information where the aspect ratio of the display is critical (e.g., circles, squares, pie-charts, etc.) it must insure that the following ratio is the same as the aspect ratio of the physical screen upon which the image is displayed.

```
(max_x - min_x) / (max_y - min_y)
```

When the turtlegraphics unit is initalized, min_x and min_y are set to 0. max_x is initialized to the number of pixels in the x direction, and max_y is initialized to the number of pixels in the y direction. In order to change to different units that still have the same aspect ratio, a call similar to the following example can be used.

```
Display_scale(0, 0, 100*ASPECT_RATIO, 100);
```

This utilizes Function Aspect_ratio described above, and makes the y axis 100 units long.

Turtlegraphics always treats the turtle as being in a fixed pixel location. Changing the scaling of the system with a call to this routine in the middle of a program does not alter the pixel position of any of the turtles in the figures. However, the values returned from X_pos and Y_pos may change.

IX.8.1.5 Figures and the Port

The programmer can create and delete new figures, each with its own turtle. When a new figure is created, it is assigned an integer, and this integer refers to that figure in subsequent calls to turtlegraphics procedures. New figures can be saved (Put_Figure) or displayed on the screen (Getfigure).

The actual display is always referred to as figure 0.

The active portion of the display can be restricted by calling viewport, which creates a "window" on the screen in which all subsequent graphics activity takes place. The user might create a figure, specify the port, then display that figure (or a portion of it) within the port. Specifying a viewport does not restrict turtle activity, it merely restricts what is displayed on the screen.

User-created figures can be saved in p-System disk files.

Function Create_Figure (x_size,y_size: real): integer

Creates a new figure that is rectangular, and has the dimensions (x_size, y_size), where (0,0) designates the lower left-hand corner. The dimensions are in units of the current display scale. The figure is identified by the integer returned by Create_figure.

When a figure is created it contains its own turtle, which is located at the initialization position or 0,0 and has a direction of 0 (it faces the right-hand side of the figure). The turtle in a user-created figure can be used by calling Activate_Turtle.

Procedure Delete_figure (screen: integer);

Discards a previously created display figure area.

Though figures may be created and destroyed, indiscriminate use of these contructs may rapidly exhaust the memory available in the p-System due to heap fragmentation. For example, a figure may be created using Create_Figure (or it may be read in from disk using Function Load_Figure, described below). If possible, after that figure is used (for example, with a Get_Figure, Put_Figure, Load_figure or Store_Figure operation) it should be deleted before other figures are created. If many figures are created, and randomly deleted, the heap fragmentation problem may occur.

**Procedure Get_Figure (source_screen: integer;
corner_x,corner_y: real; mode: integer);**

Transfers a user-created figure (the source) to the display screen (the destination) using the drawing mode specified. The figure is placed on the display such that its lower left corner is at (corner_x, corner_y). The x and y positions are specified in the units of the current display scale. If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

The following items define the drawing mode numbers.

0 Nop. Does not alter the destination.

1 Substitute. Each pixel in the source replaces the corresponding pixel in the destination.

2 Overwrite. Each pixel in the source that is not of the source's background color replaces the corresponding pixel in the destination.

3 Underwrite. Each pixel in the source that is not of the source's background color is copied to the corresponding pixel in the destination only if the corresponding pixel is of the destination's background color.

4 Complement. For each pixel in the source that is not of the source's background color, the corresponding pixel in the destination is complemented.

Values greater than 4 are treated as Nop.

If a portion of the source figure falls outside the display or the window, it is set to the source's background color.

Procedure Put_Figure (destination_screen: integer; corner_x,corner_y: real; mode: integer);

Transfers a portion of the display screen to a user-created figure using the drawing mode specified (see above). The portion transferred to the figure is the area of the display that the figure covers when it is placed on the display with its lower left-hand corner is at (corner_x, corner_y). If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

NOTE: When a figure is moved to the display by Get_Figure, further modifications to the display do not affect the copy of the figure that is saved in memory. If the user wishes to save the results of graphics work on the display, it is necessary to call Put_Figure.

Procedure Viewport (min_x,min_y,max_x,max_y: real);

Defines the boundaries of a "window" that confines subsequent graphics activities. The viewport procedure applies only to the actual display. When a window has been defined, graphics activities outside of it are neither displayed nor retained in any way. Therefore, lines, or portions thereof, that are drawn outside the window are essentially lost and will not be displayed (this is true even if the window is subsequently expanded to encompass a previously drawn line). The viewport boundaries are specified in the units of the current display scale. If the specified size of the viewport is larger than the current range of the display, the Viewport is truncated to the display limits.

IX.8.1.6 Pixels

It is possible to ascertain (Read_pixel) or alter (Set_pixel) the color of an individual pixel within a given figure. These routines are more specific than the turtle-moving routines. They are less straightforward to use, but give the programmer greater control.

Function Read_pixel (screen: integer; x,y: real): integer;

Returns the value of the color of the pixel at the x,y location in the specified figure. The x,y location is specified in the units of the current Display_scale.

Procedure Set_pixel (screen: integer; x,y: real; shade: integer);

Sets the pixel at the x,y location of the specified figure to the specified color. The x,y location is specified in the units of the current Display_scale.

IX.8.1.7 Fotofiles

The programmer may create disk files that contain turtlegraphics figures. New figures may be written to a file, and old figures restored for viewing or modification.

When figures are written to a file, they are written sequentially, and assigned an 'index' that is their location in the file. They may be retrieved "randomly" by using this index value.

The p-System name for files of figures always contains the suffix '.FOTO'. It is not necessary to use this suffix when calling Read_figure_file or Write_figure_file (if absent, it will be supplied automatically).

Function Read_figure_file (title: string): integer;

Specifies the title of a file from which all subsequent figures will be loaded. If a figure file is already open for reading when this function is called, it is closed before the new file is opened. Only one figure file may be open for reading at a single time. This function returns an integer value which is the ioresult of opening the file.

Function Write_figure_file (title: string): integer;

Creates an output file into which user-created figures may be stored. If another figure file is open for writing when this function is called, it is closed, with lock, before the new file is created. Only one figure file may be open for writing at a single time. This function returns an integer result which is the ioresult of the file creation.

Function Load_figure (index: integer): integer;

Loads the indexed figure from the current input figure file and assigns it a new, unique, figure number. An automatic Create_figure is performed. If the operation fails for any reason, a Figure_number of zero (0) is returned.

Function Store_figure (figure: integer): integer;

Sequentially writes the designated figure to the output figure file. The function returns an integer that is the figure's positional index in the current output figure file. Positional indexes start at one (1). If the index returned equals zero (0), turtlegraphics did not successfully store the figure.

IX.8.1.8 Routine Parameters

The next page shows the interface section for the turtlegraphics unit, including the parameters to all turtlegraphics routines:

```
unit turtlegraphics;

    interface

    procedure Display_scale( min_x, min_y,
                             max_x, max_y: real);
    function Aspect_ratio : real;
    function Create_figure( x_size, y_size:
                             real ) : integer;
    procedure Delete_figure( screen:
                             integer );
    procedure Viewport( min_x, min_y, max_x,
                             max_y : real );
    procedure Fillscreen( screen:
                             integer; shade:
                             integer);
    procedure Background( screen: integer;
                             shade : integer );
    function Read_pixel( screen: integer;
                             x, y : real ) :integer;
    procedure Set_pixel( screen:integer;
                             x,y:real; shade:color );
    procedure Get_Figure( source_screen:
                             integer,
                             corner_x, corner_y: real;
                             mode : integer );
    procedure Put_Figure( destination_screen:
                             integer,
                             corner_x, corner_y: real;
                             mode : integer );
    function Read_figure_file( title : string ):
                             integer;
    function Write_figure_file( title : string ):
                             integer;
    function Load_figure( index : integer ):
                             integer;
    function Store_figure( figure: integer):
                             integer;
    procedure Activate_Turtle( screen:
                             integer );
    function Turtle_x : real;
    function Turtle_y : real;
    function Turtle_angle : real;
    procedure Move( distance : real );
    procedure Moveto( x,y : real );
    procedure Turn( rotation : real );
```

```
procedure Turnto(  heading : real );
procedure Pen_mode( mode : integer );
procedure Pen_color( shade : integer );
procedure WChar( c: char; copymode, shade: integer );
procedure WString( s: string; copymode, shade: integer);
```

IX.8.1.9 Sample Program

Here is a sample program that illustrates a number of turtlegraphics routines:

```
program Spiraldemo;

  uses Turtlegraphics;

  const  nop = 0;
         substitute = 1;

  var I, J, Mode: integer;
      C: char;
      Color: integer;
      Seed: integer;
      LX, LY, UX, UY: real;

  function Random (Range: integer): integer;
    begin
      Seed:= Seed * 233 + 113;
      Random:= Seed mod Range;
      Seed:= Seed mod 256;
    end;

  procedure ClearBottom;
    {clears bottom line of screen
     for prompts}
    begin
      Penmode (nop);
      Moveto (0, 0);
      WString ('                                        ', substitute, 1);
    end;

  begin
    ClearBottom;       {various initializations}
    WString ('ENTER RANDOM NUMBER: ', substitute, 1);
    read(keyboard, Seed);
    ClearBottom;
    Display_Scale (0, 0, 200*Aspect_Ratio, 200);
```

```
    {Aspect_Ratio used so
     pattern will be round}
Color:= 0;
WString ('ENTER VIEWPORT LL CORNER: ', substitute, 1);
read(keyboard, LX,LY);
ClearBottom;
WString ('ENTER VIEWPORT UR CORNER: ', substitute, 1);
read(keyboard, UX,UY);
ClearBottom;
WString ('PENMODE = ', substitute, 1);
read(keyboard, MODE);
Fillscreen(0,1);
ViewPort (LX, LY, UX, UY);    {create port}
PenMode (0);
    {use blank pen while moving it}
Moveto (100*Aspect_Ratio, 100);
    {put turtle in center of port}
    {Aspect_Ratio ensures that it will be
     correctly centered}
PenMode (Mode);
    {set pen to selected color}
J:= Random(90)+90;
    {angle by which turtle will move
     note that turtle begins facing right
     and will move counterclockwise
     (J is positive)}

for I:= 2 to 200 do
    {draw spiral in 200 segments
     of increasing length}
    begin

          {alternate between colors}
      if Color=0 then Color := 1 else Color := 0;

      PenColor (Color);
      Move(I);
      Turn(J);
    end;

I:= Create_Figure (UX-LX, UY-LY);
    {create figure the size of the port}
PutFigure (I, LX, LY, 1);
    {save it; mode overwrites
     old figure (if any)}
```

```
    ViewPort (0, 0, Aspect_Ratio*200, 200);
        {re-specify viewport in
         the lower left corner}
    GetFigure (1, 0, 0, 1);
        {display finished spiral}
    readln;
        {clear user input buffer}
end.
```

IX.8.2 Using Turtlegraphics from FORTRAN

To use the Turtlegraphics routines from FORTRAN requires accessing special interface units at compile time because the Pascal syntax contained in the standard Turtlegraphics unit is not compatible with FORTRAN. In order to use Turtlegraphics from FORTRAN, the FORTRAN source program must contain the directive:

```
$USES TURTLEGRAPHICS IN FTN.TURTLE.CODE
```

The Turtlegraphics unit in FTN.TURTLE.CODE contains the special interface section that is FORTRAN compatible. It is accessed at compile time only. During program execution, the Turtlegraphics unit in SYSTEM.LIBRARY is automatically accessed.

In addition, there are two functions, Readfigurefile and Writefigurefile, and one procedure, Wstring, that cannot be directly referenced from FORTRAN. Instead a separate unit, FTURTLEGRAPHICS, exists to provide the support necessary to pass FORTRAN arguments to these three routines. In order to call Readfigurefile, Writefigurefile, or Wstring, the directive:

```
$USES FTURTLEGRAPHICS
```

must appear in the FORTRAN program.

To use the Turtlegraphics routines described in this chapter, these general guidelines should be obeyed:

- FORTRAN allows identifiers to be a maximum of six characters. Pascal routines with longer names need to be truncated when they are called from FORTRAN.
- Pascal boolean variables are referred to as logical in FORTRAN.
- FORTRAN refers to procedures as subroutines. The word CALL must precede the subroutine name in FORTRAN to indicate a subroutine or procedure call.

The remainder of this section describes the parameters of the routines using the appropriate FORTRAN syntax.

```
SUBROUTINE MOVE ( DISTANCE )
REAL DISTANCE
```


```
SUBROUTINE MOVETO ( X, Y )
REAL X, Y

SUBROUTINE TURN ( ROTATI )
REAL ROTATI

SUBROUTINE TURNTO ( HEADIN )
REAL HEADIN

SUBROUTINE PENCOL ( SHADE )
INTEGER SHADE

SUBROUTINE PENMOD ( MODE )
INTEGER MODE

REAL FUNCTION TURTLX ( )

REAL FUNCTION TURTLY ( )

REAL FUNCTION TURTLA ( )

SUBROUTINE ACTIVA ( SCREEN )
INTEGER SCREEN

SUBROUTINE FILLSC ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

SUBROUTINE BACKGR ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

SUBROUTINE DISPLA ( MINX, MINY, MAXX, MAXY )
REAL MINX, MINY, MAXX, MAXY

REAL FUNCTION ASPECT ( )

INTEGER FUNCTION CREATE ( XSIZE, YSIZE )
REAL XSIZE, YSIZE

SUBROUTINE DELETE ( SCREEN )
INTEGER SCREEN

SUBROUTINE GETFIG ( SOURCE, XCOR, YCOR, MODE )
INTEGER SOURCE, MODE
REAL XCOR, YCOR
```

```
SUBROUTINE PUTFIG ( DESTIN, XCOR, YCOR, MODE )
INTEGER DESTIN, MODE
REAL XCOR, YCOR

SUBROUTINE VIEWPO ( MINX, MINY, MAXX, MAXY )
REAL MINX, MINY, MAXX, MAXY

INTEGER FUNCTION READPI ( SCREEN, X, Y )
INTEGER SCREEN
REAL X, Y

SUBROUTINE SETPIX ( SCREEN, X, Y, SHADE )
INTEGER SCREEN, SHADE
REAL X, Y

INTEGER FUNCTION LOADFI ( INDEX )
INTEGER INDEX

INTEGER FUNCTION STOREF ( FIGURE )
INTEGER FIGURE

SUBROUTINE WCHAR ( C, COPYMODE, SHADE )
CHARACTER*1 C
INTEGER COPYMODE, SHADE
```

A character string cannot be passed directly to the functions Read_figure_file and Write_figure_file and to the procedure Wstring. Consequently, the FORTRAN routines FREAD, FWRITE, and FWSTRI (contained in the FORTRAN runtime interface unit FTURTLEGRAPHICS in the FORTRAN SYSTEM.LIBRARY) must be called instead.

For the functions FREADF and FWRITE, TITLE refers to the name of the fotofile that is to be read or written, and LEN contains the length of the TITLE variable.

```
INTEGER FUNCTION FREADF ( TITLE, LEN )
CHARACTER*N TITLE
INTEGER LEN

INTEGER FUNCTION FWRITE ( TITLE, LEN )
CHARACTER*N TITLE
INTEGER LEN
```

The following FORTRAN statements are necessary to call these routines:

```
CHARACTER * 5 TITLE
INTEGER LEN, IRESLT
```

```
LEN = 5
TITLE = 'FOTO1'
IRESLT = FWRITE ( TITLE, LEN )
```

or

```
IRESLT = FREADF ( TITLE, LEN )
```

In the following syntax for FWSTRI, C is actually a string of length LEN that is to be written in mode COPYMODE with shade SHADE.

```
SUBROUTINE FWSTRI ( C, LEN, COPYMODE, SHADE )
CHARACTER * <length of string> C
INTEGER LEN, COPYMODE, SHADE
```

To call FWSTRI, use the following FORTRAN statements:

```
LEN = <length of string>
C   = <string>
CALL FWSTRI ( C, LEN, COPYMODE, SHADE )
```

IX.9 Print Spooling

The print spooler is a program that allows the user to queue and print files concurrently with the normal execution of the p-System (while the console is waiting for input from the keyboard). The queue it creates is a file called *SYSTEM.SPOOLER, and the files the user wishes to print must reside on volumes that are on-line or an error will occur.

When SPOOLER is eX(ecuted, the following prompt line appears.

Spool: P(rint, D(elete, L(ist, S(uspend, R(esume, A(bort, C(lear, Q(uit

The following paragraphs define the prompt line options.

P(rint. Prompts for the name of a file to be printed. This name is then added to the queue. If SYSTEM.SPOOLER does not already exist, it is created. In the simplest case, P(rint may be used to send a single file to the printer. Up to 21 files may be placed in the print queue.

D(elete. Prompts for a file name to be taken out of the print queue. All occurrences of that file name are taken out of the queue.

L(ist. Displays the files currently in the queue.

S(uspend. Temporarily halts printing of the current file.

R(esume. Continues printing the current file after a S(uspend. R(esume also starts printing the next file in the queue after an error or an A(bort.

A(bort. Permanently stops the printing process of the current file and takes it out of the queue.

C(lear. Deletes all file names from the queue.

Q(uit. Exits the spooler utility and starts transferring files to the printer.

If an error occurs (e.g., a nonexistent file is specified in the queue), the error message appears only when the p-System is at the main system prompt line. If necessary, the spooler waits until the user returns to the outer level.

Program output to the printer may run concurrently with spooled output. The spooler finishes the current file and then turns the printer over to the user program. (The user program is suspended while it waits for the printer.) The user program should only do Pascal (or other high-level) writes to the printer. If the user program does printer output using unitwrite, the output is sent immediately and appears randomly interspersed with the spooler output.

The utility SPOOLER.CODE uses the operating system unit SPOOLOPS. Within this unit is a process called spooltask. Spooltask is started at boot time and runs concurrently with the rest of the UCSD p-System. The print spooler automatically restarts at boot time if *SYSTEM.SPOOLER is not empty. When the file *SYSTEM.SPOOLER exists, spooltask prints the files that it names. Spooltask runs as a background to the main operations of the p-System.

*SPOOLER.CODE interfaces with SPOOLOPS and uses routines within it to generate and alter the print queue within *SYSTEM.SPOOLER.

To restart the print spooling process if SPOOLER.CODE is executing when the system goes down, reboot the system, press X(ecute from the command prompt line, enter *SPOOLER.CODE, and press <return>. Then press R(esume.

X. TURNKEY APPLICATIONS FACILITIES

X.1 SYSTEM.MENU

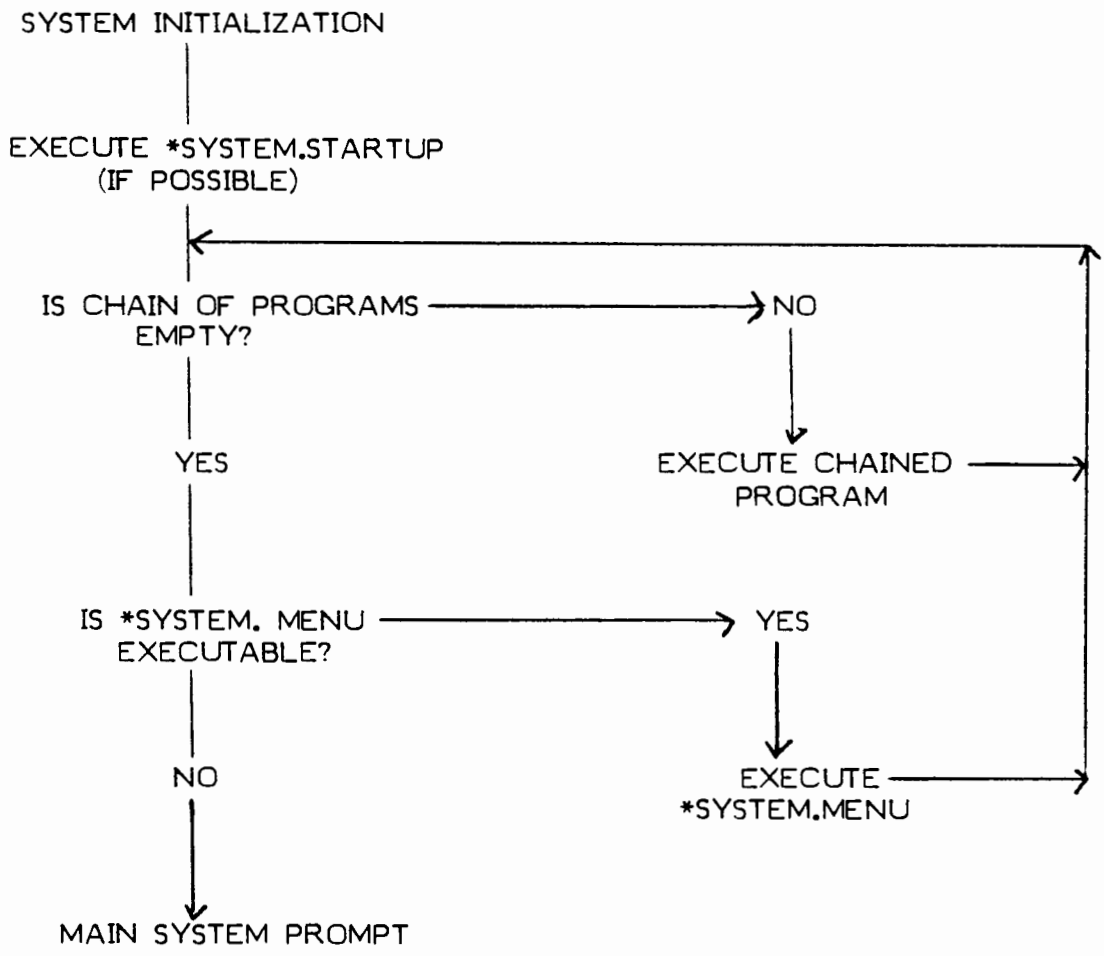
If an applications programmer wishes to design display prompts for a particular application, he or she may write a program called *SYSTEM.MENU that will display a menu for his or her particular applications environment. The main p-System prompt line need never be displayed if the program chaining facilities are used.

The user must place the compiled code file called SYSTEM.MENU on the system disk. The p-System will execute SYSTEM.MENU each time the system prompt line would normally appear.

SYSTEM.MENU works much like SYSTEM.STARTUP. If it is present, the system will execute SYSTEM.MENU any time the user initializes the system.

X.2 System Initialization

The following diagram illustrates the implementation and the flow of control within the operating system.



XI. APPENDICES

XI.A Appendix A -- Execution Errors

Execution error messages are displayed by the p-System under certain circumstances. If a code segment is needed and the disk containing it is not in the appropriate drive, the user is prompted to replace the disk and type <space> to continue. If a program attempts to divide by zero, or access outside the bounds of a Pascal array, a message indicates this and the user is prompted to type <space>, at which point the p-System is re-initialized. These and other execution errors are listed below.

0	System error	... FATAL
1	Invalid index, value out of range	
2	No segment, bad code file	
3	Procedure not present at exit time	
4	Stack overflow	
5	Integer overflow	
6	Divide by zero	
7	Invalid memory reference <bus timed out>	
8	User break	
9	System I/O error	... FATAL
10	User I/O error	
11	Unimplemented instruction	
12	Floating point math error	
13	String too long	
14	Halt, Breakpoint	
15	Bad Block	

All runtime errors cause the System to l(nitialize itself; FATAL errors cause the System to re-bootstrap. Some FATAL errors leave the System in an irreparable state, in which case the user must re-bootstrap by hand.

XI.B Appendix B -- I/O Results

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Device Number
3	Illegal I/O request
4	Data-com timeout
5	Volume is no longer on-line
6	File is no longer in directory
7	Bad file name
8	No room, insufficient space on volume
9	No such volume on-line
10	No such file on volume
11	Duplicate directory entry
12	Not closed: attempt to open an open file
13	Not open: attempt to access a closed file
14	Bad format: error in reading real or integer
15	Ring buffer overflow
16	Volume is write-protected
17	Illegal block number
18	Illegal buffer

XI.C Appendix C -- HP-86/87 Character and Key Codes

HP-86/87 display characters consist of one byte (eight bits) of information, corresponding to decimal codes 0 through 255. Characters corresponding to decimal codes 32 through 126 are standard printable characters as defined by the American Standard Code for Information Interchange (ASCII).

Mnemonic	Decimal Code	Octal Code	Keystroke	Display Character	Decimal Code	Octal Code	Keystroke
NULL	0	0	[@]c	space	32	40	space bar
SOH	1	1	[A]c	!	33	41	[!]s
STX	2	2	[B]c	"	34	42	["]s
ETX	3	3	[C]c	#	35	43	[#]s
ET	4	4	[D]c	$	36	44	[$]s
ENQ	5	5	[E]c	%	37	45	[%]s
ACK	6	6	[F]c	&	38	46	[&]s
BEL	7	7	[G]c	'	39	47	[']
BS	8	10	[H]c	(	40	50	[(]s
HT	9	11	[I]c	)	41	51	[)]s
LF	10	12	[J]c	*	42	52	[*]s
VT	11	13	[K]c	+	43	53	[+]s
FF	12	14	[L]c	,	44	54	[,]
CR	13	15	[M]c	-	45	55	[-]
SO	14	16	[N]c	.	46	56	[.]
SI	15	17	[O]c	/	47	57	[/]
DLE	16	20	[P]c	0	48	60	[0]
DC1	17	21	[Q]c	1	49	61	[1]
DC2	18	22	[R]c	2	50	62	[2]
DC3	19	23	[S]c	3	51	63	[3]
DC4	20	24	[T]c	4	52	64	[4]
NAK	21	25	[U]c	5	53	65	[5]
SYN	22	26	[V]c	6	54	66	[6]
ETB	23	27	[W]c	7	55	67	[7]
CAN	24	30	[X]c	8	56	70	[8]
EM	25	31	[Y]c	9	57	71	[9]
SUB	26	32	[Z]c	:	58	72	[:]s
ESC	27	33	[[]c	;	59	73	[;]
FS	28	34	[\]c	<	60	74	[<]s
GS	29	35	[]]c	=	61	75	[=]
RS	30	36	[^]c	>	62	76	[>]s
US	31	37	[_]c	?	63	77	[?]s

Display Character	Decimal Code	Octal Code	Keystroke	Display Character	Decimal Code	Octal Code	Keystroke
@	64	100	[@]s	`	96	140	[KEY]s [LABEL]
A	65	101	[A]s	a	97	141	[A]
B	66	102	[B]s	b	98	142	[B]
C	67	103	[C]s	c	99	143	[C]
D	68	104	[D]s	d	100	144	[D]
E	69	105	[E]s	e	101	145	[E]
F	70	106	[F]s	f	102	146	[F]
G	71	107	[G]s	g	103	147	[G]
H	72	110	[H]s	h	104	150	[H]
I	73	111	[I]s	i	105	151	[I]
J	74	112	[J]s	j	106	152	[J]
K	75	113	[K]s	k	107	153	[K]
L	76	114	[L]s	l	108	154	[L]
M	77	115	[M]s	m	109	155	[M]
N	78	116	[N]s	n	110	156	[N]
O	79	117	[O]s	o	111	157	[O]
P	80	120	[P]s	p	112	160	[P]
Q	81	121	[Q]s	q	113	161	[Q]
R	82	122	[R]s	r	114	162	[R]
S	83	123	[S]s	s	115	163	[S]
T	84	124	[T]s	t	116	164	[T]
U	85	125	[U]s	u	117	165	[U]
V	86	126	[V]s	v	118	166	[V]
W	87	127	[W]s	w	119	167	[W]
X	88	130	[X]s	x	120	170	[X]
Y	89	131	[Y]s	y	121	171	[Y]
Z	90	132	[Z]s	z	122	172	[Z]
[	91	133	[[]s	{	123	173	[/]sn
\	92	134	[\]	|	124	174	[|]sn
]	93	135	[]]s	}	125	175	[-]sn
^	94	136	[^]s	~	126	176	[*]sn
_	95	137	[_]s	(DEL)	127	177	[+]sn

c Indicates that the [CTRL] key is held down while the letter or symbol key is pressed.

n On the numeric keypad.

s Indiates that [SHIFT] is held down while the letter or symbol key is pressed. (The assumption is that [CAPS LOCK] key is in the up position.)

Inverse video characters are displayed by setting the eighth--or most significant--bit. To do so, add 128 decimal to the character code of the display character. For example, to display a highlighted "A" from Pascal, use "WRITE(CHR(ORD('A')+128));".

The key codes assigned to individual HP-86/87 keys enable you to control the keyboard during program execution. In Pascal, you can use "READ(KEYBOARD,Ch)" statements and test the returned values against the codes listed in this appendix. For example, if ORD(Ch) = 65, then the user has pressed shifted [A].

Refer also to the Pascal Reference Manual for information concerning the differences between READ and READLN statements. For example, if [-LINE] is pressed in response to a "READ(Ch)" statement, the System is passed character code 157; if, however, [-LINE] is pressed during the execution of a "READLN(Str)" statement, its definition in SYSTEM.MISCINFO as the KEY TO DELETE LINE will be used.

During an executing program, the System can be immediately interrupted by the [SHIFT][RESET] keystroke or by the KEY FOR BREAK, initially [CTRL][P].

Key	Decimal Code	Octal Code	Key	Decimal Code	Octal Code
[TR/NORM]s	27	33	[KEY LABEL]	150	226
[k1]	128	200	[↖]s	152	230
[k2]	129	201	[BACK SPACE]	153	231
[k3]	130	202	[BACK SPACE]s	155	233
[k4]	131	203	[k7]	156	234
[k8]s	132	204	[-LINE]	157	235
[k9]s	133	205	[I/R]s	158	236
[k10]s	134	206	[<--]	159	237
[k11]s	135	207	[E] n	160	240
[-CHAR]s	136	210	[k5]	161	241
[CLEAR]s	137	211	[k6]	162	242
[INIT]s	140	214	[↑]	163	243
[RUN]	141	215	[↓]	164	244
[CONT]	143	217	[k12]s	165	245
[STEP]	144	220	[RESULT]s	166	246
[ROLL ^]s	145	221	[A/G]s	168	250
[TEST]s	146	222	[ROLL v]	169	251
[k14]s	147	223	[-->]	170	252
[PLIST]s	149	225	[k13]s	172	254

XI.D Appendix D -- Using an RS-232C Serial Interface

Serial I/O is available through the addition of serial interface cards. Four devices may be controlled through the use of four serial cards plugged into the back of the HP-86/87. The p-System will automatically recognize their presence, setting the baud rate to 300, parity to NONE, and bits per character to 8. They may be accessed as volumes REMIN:/REMOUT: (or SERIALA:), SERIALB:, SERIALC:, and SERIALD:, or through the p-System device number. The device number of the first serial device (lowest select code) is determined by adding the MAX NUMBER OF SUBSIDIARY VOLS (10) to the FIRST SUBSIDIARY VOL NUMBER (24). For example, REMIN:, REMOUT:, SERIALA:, and device #34: all specify the same serial interface.

The baud rate, parity, and bits per character may be changed by using the utility SETSERIAL.CODE. Most of the serial interface registers may be written to using this utility. Refer to the HP 82939A Serial Interface Owner's Manual for details about the registers.

The utility REMTALK.CODE can be used to transfer p-System files between two p-System machines that are equipped with compatible serial interfaces that have been connected together. (If any of the switch settings on the serial interfaces have been changed, it is a good idea to restore them to the factory settings.) REMTALK.CODE should be run on each system simultaneously.

When REMTALK.CODE is executed, the following prompt will appear:

```
REMTALK [IV.0 a1] - press S(lave first
M(aster  S(lave  Q(uit
```

Designate one System as the slave and the other System as the master by pressing [S] first on the one and then [M] on the other. The order is important. Once the slave-master relationship is established, the master will be used to control the file transfer. The master will display:

```
S(end  R(eceive  Q(uit
```

If you want to send a file from the master to the slave, press [S]. The master will prompt:

```
Send what file?
```

Type the source filename and, optionally, a volume name or device number. The master will then ask:

```
Send to what remote file?
```

Type the destination filename and, optionally, specify the slave disc on which it should reside. (You may escape either filename prompt by pressing [END LINE] alone in response to the prompt.)

Example:

```
M(aster  S(lave Q(uit M
  S(end  R(eceive Q(uit S
  Send what file? #4:READY.CODE
  Send to what remote file? #5:READY.CODE
```

As the file is being sent, the displays will show a period (.) for each block sent. When the file transfer is complete, the master will again display:

```
S(end  R(eceive  Q(uit
```

Either transfer another file from (or to) the master, or press [Q] to quit. Both displays will then return to the original prompt:

```
M(aster  S(lave  Q(uit
```

You can then exit the program or reverse System roles.

Note that it may be necessary to execute the S(end command more than once due to synchronization problems.

XI.E Appendix E -- Assembler Syntax Errors

1: Undefined label
2: Operand out of range
3: Must have procedure name
4: Number of parameters expected
5: Extra symbols on source line
6: Input line over 80 characters
7: Unmatched conditional assembly directive
8: Must be declared in .ASECT before use
9: Identifier previously declared
10: Improper format
11: Illegal character in text
12: Must .EQU before use if not to a label
13: Macro identifier expected
14: Code file too large
15: Backward .ORG not allowed
16: Identifier expected
17: Constant expected
18: Invalid structure
19: Extra special symbol
20: Branch too far
21: LC - relative to externals not allowed
22: Illegal macro paramter index
23: Illegal macro parameter
24: Operand not absolute
25: Illegal use of special symbols
26: Ill-formed expression
27: Not enough operands
28: LC - relative to absolutes unrelocatable
29: Constant overflow
30: Illegal decimal constant
31: Illegal octal constant
32: Illegal binary constant
33: Invalid key word
34: Unmatched macro definition directive
35: Include files may not be nested
36: Unexpected end of input
37: .INCLUDE not allowed in macro
38: Only labels & comments may occupy column one
39: Label expected
40: Local label stack overflow
41: String constants must be on single line
42: String constant exceeds 80 characters
43: Cannot handle this relocate count

44: No local label in .ASECT
45: Expected key word
46: String expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: Undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name directory
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definition not allowed
66: '=' or '<>' expected
67: May not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: Too many procedures
71: Only absolute expressions in .ASECT
72: Must be label expressions in .ASECT
73: No operands allowed in .ASECT
74: Offset not word-aligned
75: LC not word-aligned
76: ARP/DRP operand not register # 0 to 63
77: Address expected
78: Register 1 named (warning)
79: Literal expected after '='
80: Invalid expression for data register
81: Missing first (data register) operand
82: Missing second operand
83: Invalid expression for address register
84: Comma expected between operands
85: Invalid second operand address mode
86: # immediate bytes doesn't agree with dr
(also given if multiple bytes with byte mode instruction)
87: # immediate bytes not verified
(R# or *R used in data register field)
88: Missing jump address

XI.F Appendix F -- Summary of Differences Between Versions

1. History

The UCSD p-System Version IV.1 (its latest release) has undergone a number of incarnations since its first release to the public. The names it has borne are: I.3, I.4, I.5, II.0, II.1, III.0, and IV.0. Most changes to the System have expanded its capabilities. The single-user microprocessor environment, portable code, and hierarchical operating system are features of the design which have not changed. Increasing capabilities has led to a proliferation and diversification of features -- this trend has been countered by efforts for standardization and portable code.

Release IV.0 was designed to incorporate the capabilities of II.0, II.1, and III.0, while cleaning up some rough edges of the user interface, UCSD Pascal code, and System internals. Release IV.1 offers a variety of System enhancements, including extended memory, turtlegraphics, file management units, and print spooling.

Before detailing new changes, here is a bit of history (may be skipped by the eager):

After a series of releases internal to UCSD and its computer science program, I.3 was made available to the general public. It was a very simple and very stable version of the System. Though a screen-oriented editor had existed for some time, I.3's System editor was YALOE. I.3 ran on PDP-11's and LSI-11's.

I.4 was the first version to be available on other microprocessors, including 8080's and Z80's running CP/M. I.4 also introduced the full Screen Oriented Editor.

I.5 introduced separate compilation and assembly. External routines and UNITs could be bound into host programs with the Linker. Still more microprocessors were supported.

II.0 was essentially a stabler version of I.5. It was released by UCSD shortly before SofTech Microsystems assumed responsibility for its licensing and support.

II.1 is the variety of II.0 distributed by Apple Computer Corp. It has the INTRINSIC UNIT feature, and a number of minor differences.

III.0 is distributed by Western Digital Corp. To run UCSD Pascal on a hardware-emulated P-machine required many changes, mostly internal. At the level of Pascal object code, III.0 introduced concurrent procedures called processes.

IV.0 pulls together the user-level features of the last three versions.

IV.1 is new, and includes the features that have accrued since the release of IV.0.

2. Version IV.0

1. Media -- the logical format of disk directories and disk files has not changed, so no conversion of text or data is required.

2. Source Code -- Pascal and FORTRAN source from versions II.0, II.1, and III.0 will compile under IV.0 . Most programs will then run. Those that will not are programs dependent on former implementations of the System's data structures and memory management, or possibly on the memory requirements of a given machine (i.e., "tight-fitting" programs).

3. Object Code -- old programs must be recompiled. The byte sex of a host processor no longer matters -- it is detected and properly dealt with by the Operating System. FLIPCODE and FLIPDIR are no longer needed.

4. Pascal -- has been extended with the PROCESS construct for concurrency. SEPARATE UNITS and INTRINSIC UNITS are no more, although they will still be compiled as regular UNITs. UNITs need not be bound in by Linker and therefore may be shared (i.e., they behave as II.1 INTRINSIC UNITs but are not bound to a single segment number). The IMPLEMENTATION part of a UNIT may contain SEGMENT PROCEDUREs. A program may reference up to 256 compilation units, a compilation unit may reference up to 256 segments, and may contain up to 16 segments.

5. FORTRAN and BASIC are now part of the System.

6. The Editors -- in YALOE, the E(rase command is gone; otherwise it is unchanged. The Screen Oriented Editor remains much the same; eX(change is more flexible, and a K(olumn command has been added.

7. The Assemblers -- no macro parameters are allowed within ASCII strings, the radix switch characters have changed, alphabetic alternatives to some special characters are provided, relocatable procedures have been added. Old assembly language procedures which use type STRING, and old assembly language FUNCTIONs require some changes to run under IV.0.

8. Memory Management -- SEGMENT routines may be declared (as before). A compilation module (program or UNIT) may contain up to 16 segments. The bodies of all segment routines must be declared before the bodies of any non-segment routines are declared. The standard Pascal intrinsics NEW and DISPOSE are now implemented. UCSD intrinsics MEMLOCK and MEMSWAP, and VARAVAIL, VARNEW, and VARDISPOSE have been added.

9. External Compilation -- there is now only one type of UNIT. INTRINSIC and SEPARATE UNITs which exist in old programs will be compiled into regular IV.0

UNITs. A IV.0 UNIT is like an old II.1 INTRINSIC UNIT in that it need not be linked, and may be shared, but unlike an INTRINSIC UNIT in that it does NOT have a fixed segment number. UNITs may now contain SEGMENT routines (they must be declared in the IMPLEMENTATION part).

10. Concurrency -- is as in Version III.0. The user may declare a PROCESS which is declared like a procedure, but is started by the UCSD intrinsic START. Once a process is START'ed, it appears to run simultaneously with the host program and (possibly) other processes, until it has completed. The predeclared type SEMAPHORE has been introduced to aid in process synchronization; SEMAPHOREs can be manipulated with the intrinsics SIGNAL and WAIT.

11. Internals -- the P-codes have been slightly modified, and runtime memory management has changed. Rather than being placed on the Stack, procedure code now resides in a "code pool" which resides between the Stack and the Heap, and is relocatable. The code pool is a highly flexible structure, and allows for much runtime swapping. In addition, the following UCSD intrinsics have been created to aid in system-level memory management: MEMLOCK, MEMSWAP, VARAVAIL, VARNEW, VARDISPOSE.

12. Disk Swapping -- since code is swapped more frequently in IV.0, a number of prompts have been added which request that the user insert a needed volume.

13. Incompatibilities -- the following practices (which run under II.0, II.1, or III.0) require modification before a program can run under Version IV.1:

System Data Structure Dependencies

Many System data structures have changed. Therefore, programs which directly access such things as SYSCOM, SIB's, etc., will have to be modified -- refer to internal documentation.

Heap Storage Utilization

A program cannot assume that the memory immediately following that obtained by a NEW is unoccupied and available.

Similarly, consecutive calls to NEW do not necessarily yield a contiguous area of memory. The practice of indexing across the boundary separating storage obtained by consecutive calls to NEW will fail under Version IV.0.

Calls to MARK and RELEASE MUST be paired correctly. The pointer value obtained by calling MARK must NOT be modified prior to

calling RELEASE. Furthermore, the pointer obtained from MARK cannot be used as a base pointer for storage references.

Tightly Fitting Programs

IV.0 in general uses more memory at runtime than previous versions, so programs that have been tailored to fit in main memory will possibly need to be tailored some more. The improved memory management in IV.0 should make this an easier task than it has been in the past.

3. Version IV.1

1. Extended Memory -- It is now possible to place the code pool outside the stack/heap area within what is called "extended memory." The extended memory area may be as large as 64K bytes (chapter 5, Introduction to the p-System).

2. Editor Enhancements -- The Editor now allows user-defined tab stops and includes additional information in the S(et E(nvironment display (chapter 4, Operating System Reference Manual).

3. New Filer Capabilities (chapter 3, Operating System Reference Manual) --

The V(ols command has been upgraded to display additional information that indicates the size, in blocks, of the discs that are on-line.

The O(n/off-line command enables the creation and use of "subsidiary volumes," a file organization tool useful for large storage devices such as Winchester hard discs. Subsidiary volumes are a logical division of physical discs into a two-level file hierarchy; they can increase the number of files that may be stored on a disc.

The F(lip-swap/lock command allows memlocking of the Filer's code segments, a convenient mechanism that allows the user to remove the disc containing the SYSTEM.FILER (and the System disc, if they are different) while the Filer is being used.

4. File Management Units -- Pascal programs may now simulate the functions of the Filer using four units included in SYSTEM.LIBRARY (chapter 4, Pascal Reference Manual).

5. Pascal Compiler Enhancements -- The selective USES statement specifies to the Compiler which routines from a unit are needed. The Compiler then selects only the relevant identifiers, thereby reducing compiler-time symbol table space requirements. The selective USES statement is designed for compiling programs that use units with very large interface sections, only a small portion of which may be required (chapter 3, Pascal Reference Manual).

6. Symbolic Debugger -- The p-Code debugger for Pascal programs has been upgraded to allow the specification of break point locations with a line number instead of a p-Code offset. Also, variables in Pascal programs can be specified by name rather than by data offset (chapter 9, Operating System Reference Manual).

7. Print Spooling -- The print spooler (SPOOLER.CODE shipped on the Utilities disc) enables files to be sent to the printer during regular use of the p-System. A user may, for example, print files and use the Editor at the same time (chapter 9, Operating System Reference Manual).

8. Echoed Characters -- The user may now specify the characters that the keyboard will echo to the screen. Besides the standard ASCII character codes (0 through 127), decimal codes 128 through 255 may also be echoed if desired (chapter 3, Installation Guide).

9. Turnkey Applications Facility -- The p-System will now recognize a user-written executable code file called SYSTEM.MENU. This file is executed automatically whenever the System command promptline is normally displayed. This facility is useful for creating applications environments in which users will not be aware of the underlying p-System environment (chapter 10, Operating System Reference Manual).

10. Performance Monitor -- It is now possible to write a unit called PERFOPS and bind it into the operating system. This unit allows the user to be cognizant of p-System activities such as fault handling and segment memswapping.

INDEX

Note: In the following index, IG refers to the Installation Guide, and AG to the Internal Architecture Guide. Users interested in certain topics should refer to those manuals as well. Users interested in FORTRAN or Pascal should refer to the appropriate language reference manual.

Boldface indicates the principal description of an item.


HEWLETT
PACKARD
Personal Computer Division
1010 N.E. Circle Blvd., Corvallis, OR 97330
Reorder Number
00087-90381
©1982, SofTech Microsystems
Printed in U.S.A. 12/82
00087-90382