SAGE
COMPUTER ▊ TECHNOLOGY

p-System

Editors

Pascal

Assembler

DISCLAIMER:

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

PREFACE

"Toto, I've a feeling we're not in Kansas any more." -- The Wizard of Oz

The many incarnations of the "1st edition" of this manual reflect the growth of the UCSD p-System. In this edition, we have tried to unify the material, to motivate it and supply some background, to be a bit more supportive of the novice user. At the same time, we have tried to be as concise as possible. Our various aims have sometimes been in conflict, and we have resolved them as best we could. It is our hope that this new manual is much clearer and more usable than its predecessors.

IV.0 itself is a unified version of our System -- somewhat simpler, and a great deal more powerful. Separate compilation is easier than before, and concurrent processes are now a reality. Watching IV.0 develop has been enjoyable; hopefully this manual does it justice.

This is a Users' manual. Internal information has been banished to a different document. That should make it easier to use for the majority of users; systems users will benefit from the greater detail included in the Internal Architecture Guide.

A lot of things have been maturing: this book, the System, the community of Pascal and p-System users, the international standards for Pascal. Even though the task is endless, the creators of this system and this document will attempt to keep pace with all of these things. In this light, our System and its documentation are one half of an ongoing dialog -- your response is the other half; your opinions are not only welcome but needed.

We are no longer where we were when we started. Thank you for joining us in our ventures and explorations, and thank you for contributing your own time and your own discoveries.

R.C.
San Diego
4 November 1980

TABLE OF CONTENTS

III   FILES AND FILE HANDLING

IV THE SCREEN ORIENTED EDITOR

V YALOE -- YET ANOTHER LINE ORIENTED EDITOR   ("yah-loo-ee")

VI THE UCSD PASCAL LANGUAGE

IX CONCURRENT PROC

X UTILITIES

XI APPENDICES

1. INTRODUCTION

1.1 How to use this Manual

This is the basic reference to the UCSD p-System and UCSD Pascal. 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 Ken Bowles' Beginners Guide for the UCSD Pascal System. If you have not reached the point of actually bringing your system up (what we call "bootstrapping" it) you should consult the Installation Guide, or the documentation that is provided with your particular hardware.

Although this Manual is not a tutorial, this introduction is designed as a description of the overall structure of our System, and you should read it before doing any extensive work. Once you have worked through Bowles' book 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 chapters on Pascal, the Adaptable Assemblers, 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 VIII: 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 FORTRAN and BASIC, 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.

1

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

1.2   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.

1.2.1   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 11, 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 the Users' Manual, and if you are interested in it, you should consult the Installation Guide.

Bringing up the System starts the System's Emulator (also called the Interpreter) which executes all programs written in high-level language. While the Adaptable Assemblers generate machine language for particular target machines, the System's high-level languages such as Pascal, FORTRAN, and BASIC 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 11.

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.

FIGURE 1

FIGURE 2

6
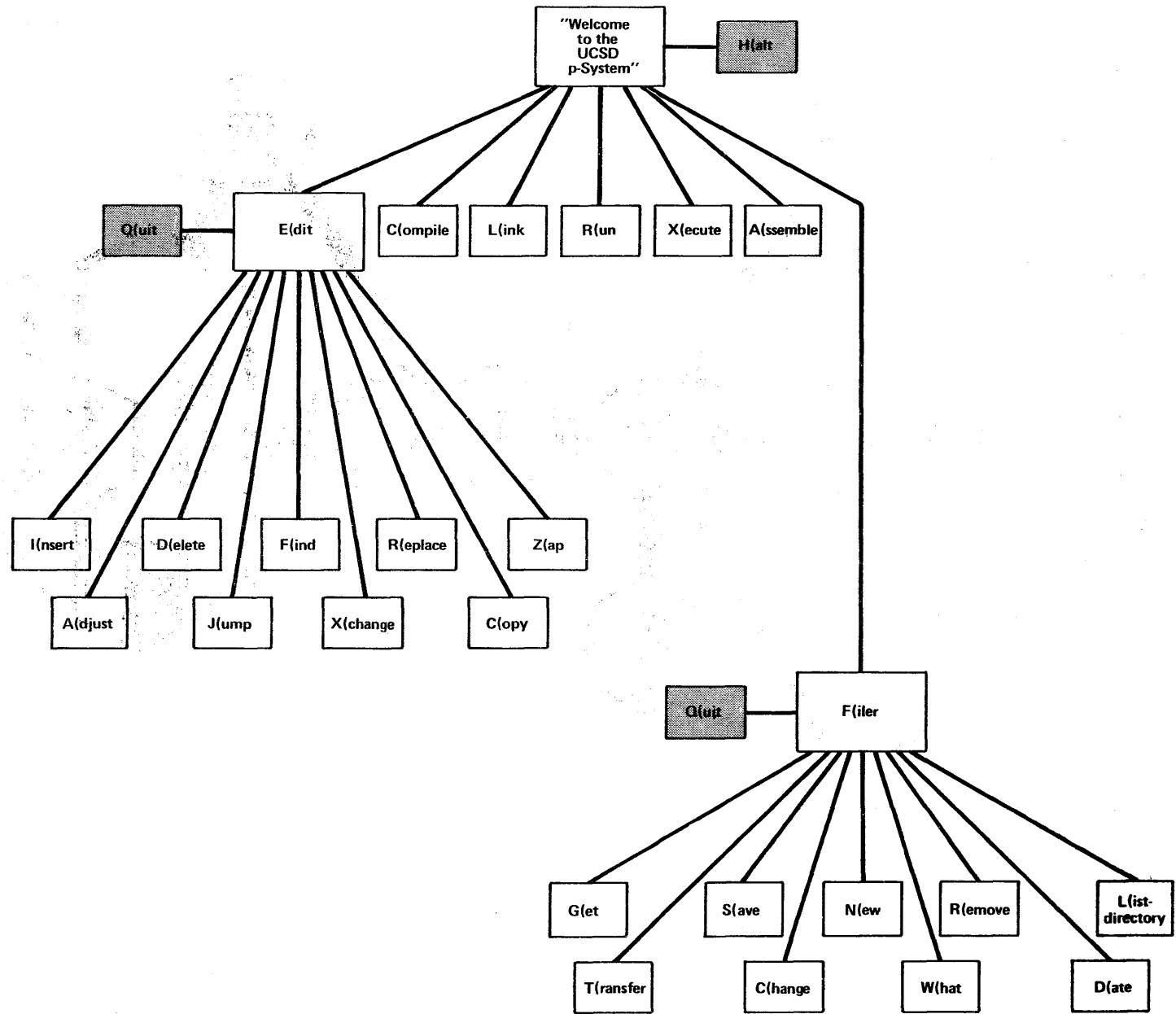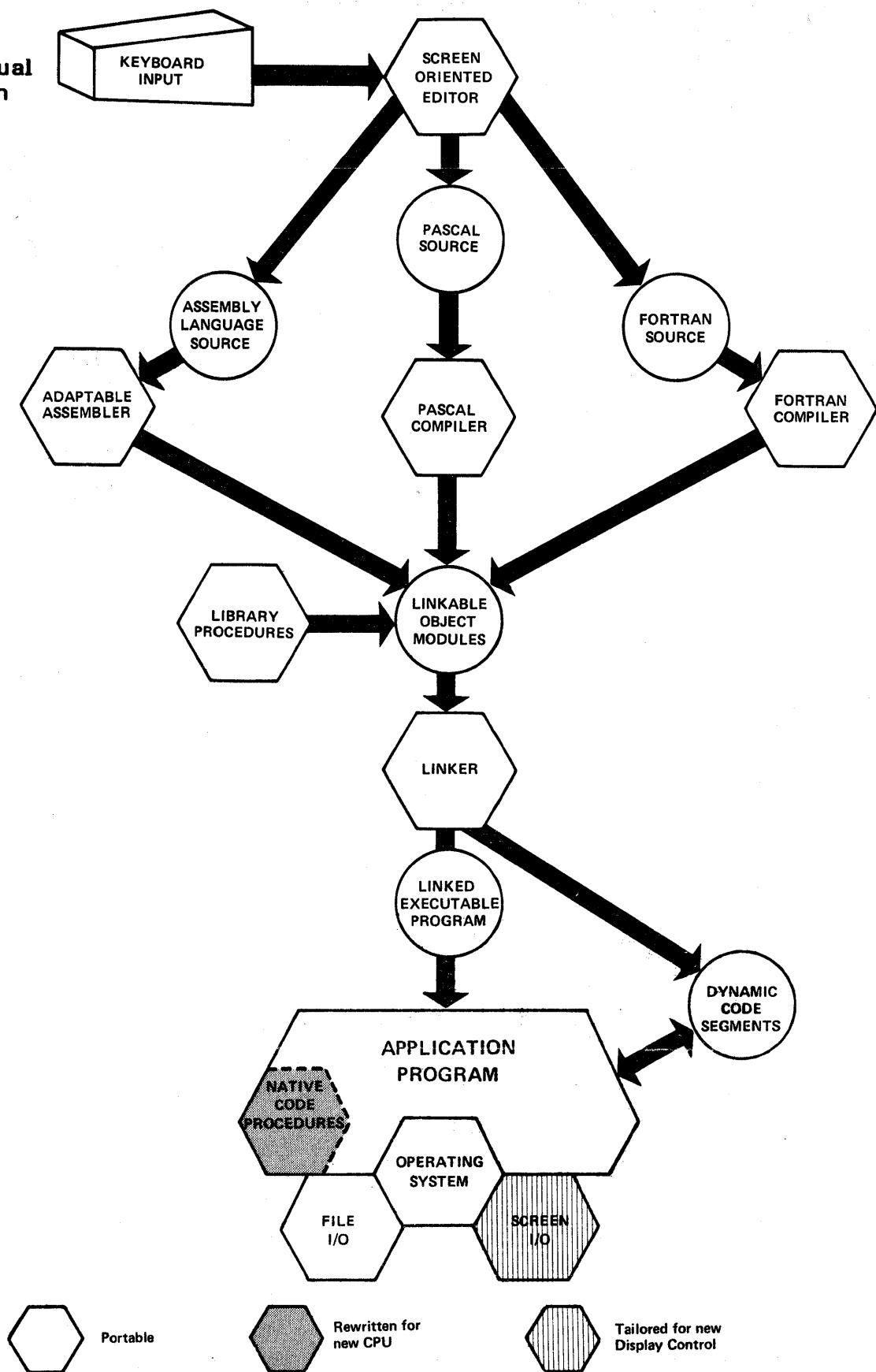
1.2.2   File Organization

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", "default disk", or "bootstrap 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 bootstrap 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 Installation Guide. This section gives a general description of the names of the major pieces of the System.

On some implementations of the p-System, many System files will reside in ROM (Read Only Memory) rather than on any disk.

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, FORTRAN, or BASIC). 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.

The emulators have various names, usually machine-specific. The most widely distributed ones are ...

SYSTEM.PDP-11 and
SYSTEM.MICRO

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 (or in ROM, for ROM-based implementations). 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, such as a PDP-11, 6502, or Z-80, to name a few. .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 <u>do</u> 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.

Other suffixes you may encounter include .BACK files, which are backups of .TEXT files, and .BAD files, which are 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.

The ways in which you can use a workfile will become more apparent from using Ken Bowles' Beginner's Guide, reading the chapters on System commands and the Filer, and of course, playing with the System yourself.

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-block-structured devices. Block-structured devices are usually disks. They contain removable 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
3	GRAPHIC:	the screen when used for graphics
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..12	<disk name>:	additional disk drives

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).

13

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, and the UCSD intrinsics REDIRECT, EXCEPTION, and CHAIN (in Chapter VI).

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
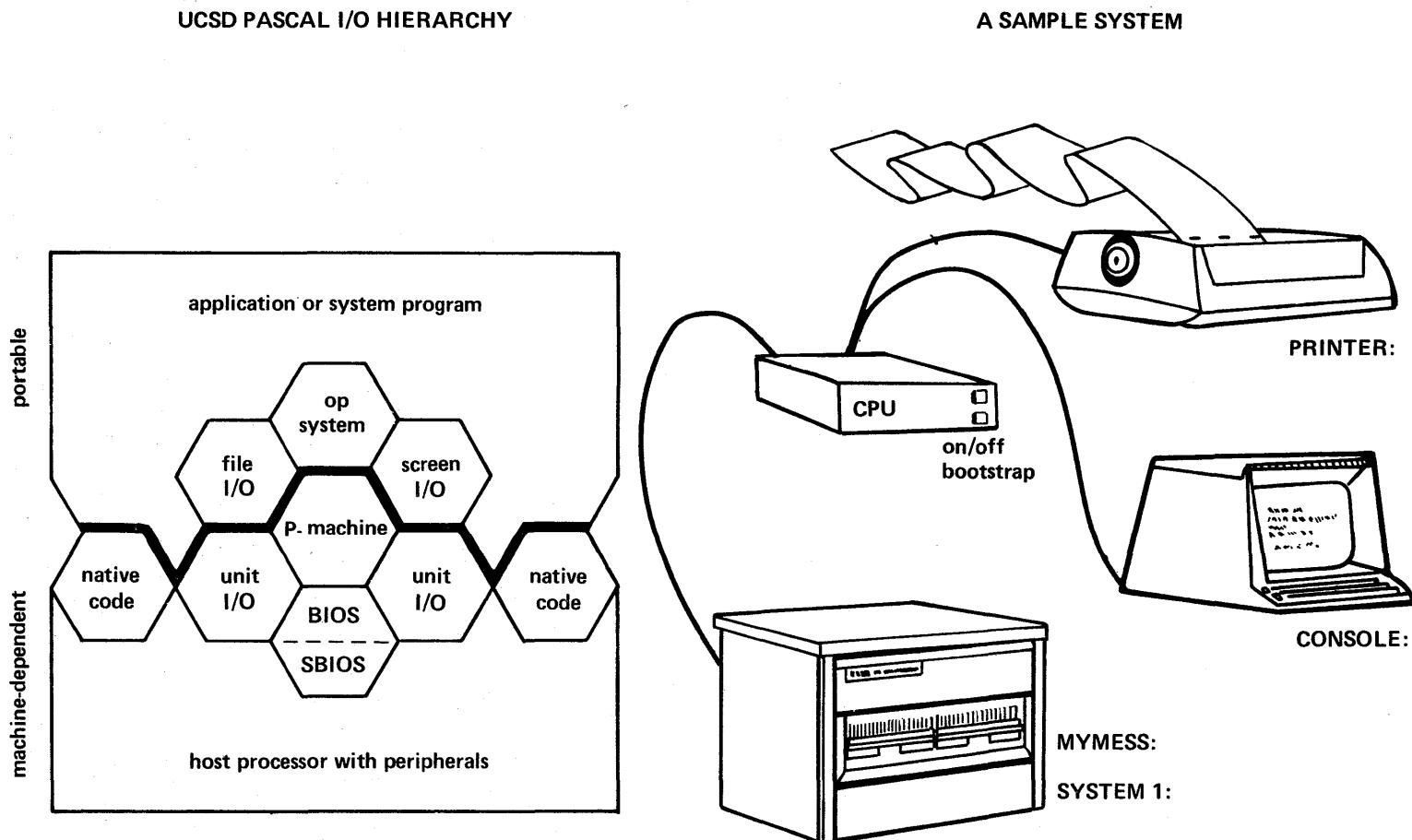
FIGURE 3

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 <u>not</u> considered subsidiary segments, but <u>are</u> 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 11). A compilation unit can also specify the library it needs by using the $U Compiler option (see Section VI.3). Libraries are discussed in detail in Chapter VIII.

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 VII.

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 Chapter VI, 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 Chapter VI).

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

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

1.3   Note on Bug Reporting

Reporting problems is a practice that benefits everyone.   Customers can learn that the problem or bug has already been solved, and what the fix is, or that it was previously unknown, and that steps will be taken to fix it in future versions. Software authors benefit from the reports -- not everyone is familiar with all the problems which users discover, nor all the applications for which the System might be used.   New uses lead to new problems, which lead in turn to new improvements.

Some users try to fix problems on their own, without consulting their supplier. There is no need to be a martyr -- we ask that you do report problems, even if you think they may already be known (it's not necessarily true), or if you have found some private solution (the solution you find may be something your supplier would like to know).

What is required in a bug report?   A phone call, or letter, or a Problem Report mailed to the appropriate support office, may all be adequate, but only if they contain certain information.   Bug reports are like UFO sightings: one report with no evidence is regarded with suspicion, many reports with no evidence will nevertheless spark an investigation, and a single report which contains evidence and a thorough description will be believed and closely pursued.

There is a pragmatic difference between a "bug" and a "glitch" -- a bug is dependable, a glitch is intermittant.   If a problem can be duplicated, then it is a bug, and much more time which is much more productive will be spent on tracking it down.   That is why we encourage you to send thorough problem reports.

We do ask that you be aware of the difference between a bug report and a

suggestion.   Some people will inevitably object to things that are intended "features" of the System.   There is nothing wrong with that -- the design process itself involves debate and compromise.   If you have a suggestion, please report it -- only through feedback can the System improve.   But please don't claim that your suggestion reports a bug -- that only confuses the issue.   Your standard here is the Users' Manual.   It attempts to describe the p-System that is sent out.   If there are discrepancies between the manual and your software, then you should submit a problem report.   If the manual accurately describes the situation you object to, then report your dissatisfaction, but realize that the way the System operates is already known.

When you report a problem, the rule of thumb is: the more information, the better.   These are the things that should be specifically stated:

environment: what part of the system was running?
             what version of Pascal were you using?
             what processor do you use?

actions: what were you trying to do?
      what were you doing immediately before
         the problem appeared?
      what exactly happened that was a problem?
      ... and in what order?

reactions: have you figured out a workaround?
      how seriously does the problem affect your work?
      have you had this problem before (even transiently)?

If you think it would help, you might include a listing with your report. Sometimes a listing will be needed to understand a problem.

Remember that debugging is the slowest part of any software development, so do not expect problems to disappear overnight. Nonetheless, we fully appreciate the time you take to fill out a useful report. Your concern for the System is what keeps it maturing.

Details of who to contact for support assistance should be included with the System you receive. If you receive your p-System through some supplier other than SofTech Microsystems, then you should always contact that supplier directly, unless you have been specifically instructed otherwise.

11.   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.

11.1 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, ? [IV.0]

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

>Edit: A(djust C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap [E.6]

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:

Filer: B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(ols, X(amine, Z(ero

... is the remainder of the Filer's menu.   (This feature is particularly useful if your console's screen is less than 80 characters wide.)

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.

You may have already seen the System promptline and screen as it appears after booting, and may have played with the System on your own or in conjunction with Ken Bowles' book. This is all the familiarity with promptlines that you need in order to start productively using the various System commands.

If you are using a line-oriented terminal (such as a teletype), then you can set HAS SLOW TERMINAL to True in your SYSTEM.MISCINFO file, and the System will abbreviate most promptlines. If you have a CRT with narrow lines (less than 80 characters wide), then set SCREENWIDTH in SYSTEM.MISCINFO to the appropriate value. The System will display as much of a promptline as will fit on a line, and typing '?' causes it to display the remaining commands. Refer to the Installation Guide for details of modifying SYSTEM.MISCINFO with the SETUP utility.

ll.2 Disk Swapping

Since the lV.0 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:

        Segment not found on device # 5
        Please replace volume USER1
        type <space> to continue

... 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:

        Please replace 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 <u>not</u> done for explicit UNITREADs and UNITWRITEs that may appear in a user program.

11.3   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 1.2.3), and changing the default library text file (see Section 1.2.4 or Chapter VIII).   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.   Causing redirection from within a user program requires the use of the REDIRECT intrinsic (and possibly the EXCEPTION intrinsic); refer to the descriptions of these intrinsics in Chapter VI.   The intrinsic CHAIN also makes use of execution option strings.

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 11.4.8.

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 VIII.

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).

ll.3.2 Redirection

The following execution option strings control redirection:

  Pl=<filename>
  Pl=<string>
  PO=<filename>
  l=<filename>
  l=<string>
  O=<filename>

Pl= redirects program input. Pl=<filename> causes the input to a program to come from the file named. Pl=<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.

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

l= redirects System input. l=<filename> causes System input to come from the file named. l=<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 Pl=, l= overrides any previous l=, and like PO=, O= overrides any previous O=. Using l= without a filename resets System input to CONSOLE:. Using O= without a filename resets System output to CONSOLE:.

For Pl=<filename> and l=<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 Pl=<string> and l=<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 <u>both</u> a file and the appropriate scratch input buffer, but if this is the case, the scratch buffer will be used <u>first</u> (until it is empty). Strings are always <u>appended</u> 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 Halt or a runtime error. An ordinary I(nitialize will not alter System redirection. The System's scratch input buffer is lost.

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). It affects file-level operations and intrinsics, but <u>not</u> device-level intrinsics such as UNITREAD, UNITWRITE, BLOCKREAD, BLOCKWRITE, and so on. It also cannot affect calls of the form:

    REWRITE(MY_FILE,'CONSOLE:');
    WRITE(MY_FILE, LOTS_OF_TEXT)

... and so forth, because these calls do not involve the standard input and output files.

A user program can also take advantage of redirection with the intrinsic REDIRECT, and clear redirection with the intrinsic EXCEPTION. The CHAIN intrinsic allows the user to "queue" an execution option string for execution <u>after</u> the program that contains it has finished executing. All these routines are described in Chapter VI.

Examples:

    Execute what file? <u>YEEN   P1=IN PO=OUT</u>

    Execute what file? <u>PO= OUT   P1= IN   yeen</u>

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

    Execute what file? <u>1=</u>

... stops System input redirection.

    Execute what file? <u>PO= storeme.text   P1= 1="fgRUNME,qr" P=WORK2</u>

... 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? <u>PO= storeme.text   P1= 1="fpWORK2:,gRUNME,qr"</u>

... 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 file? <u>GRISWICH</u>
    Code file name? <u>$</u>

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

Device names may also be used.

Example:

    Listing file? <u>PRINTER:</u>

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).

ll.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 a given native code (depending on which of the assemblers has been named SYSTEM.ASSMBLER). 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 *xxxx.ERRORS is present, where xxxx is the correct processor name, e.g., Z80.ERRORS) 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 VIl describes the Assemblers in detail.

COMPILE

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 if the file *SYSTEM.SYNTAX is present, an error message will be displayed.

Chapter VI describes the Pascal Compiler in detail. FORTRAN and BASIC are described in separate manuals.

11.4.3 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.

11.4.4 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 111.

11.4.5   HALT

On the promptline: H(alt.

Causes the System to stop execution.   On some implementations, follows this halt with a bootstrap.   On most implementations, the only way to restart the System after a H(alt is by doing a hardware bootstrap.

11.4.6   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.

A SYSTEM.STARTUP may come with the System; if it does, it gives advice on getting started; once you have followed the advice, you may delete the SYSTEM.STARTUP (possibly saving a copy for future reference).

You may also 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.

An I(nitialize will not clear any redirections (see Section 11.3), but any runtime error <u>will</u>.

11.4.7   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 VIII, particularly Section VIII.4.

11.4.8   MONITOR

On the promptline: M(on.

Redirecting the System's input (see Section 11.3) 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 1= 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, saves the monitor file, and returns the user to the System promptline.   If no monitor file is open, an error message is displayed.

A(bort ends a monitor and returns to the System promptline, 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. If monitoring had not been S(uspend'ed, nothing will happen.

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.

11.4.9 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.

USER RESTART

11.4.10   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.

11.4.11   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 11.3, above).   In the simplest case, this string contains nothing but the name of a codefile to be executed (as described in Section 11.4.0).

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 11.3, 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.

lll. FILES AND FILE HANDLING

lll.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.

lll.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> (ASCll CR), and the first two may be a blank-compression pair. The optional blank-compression pair consists of an ASCll 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.

45

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

.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;
    var fotofile: packed file of screen;

... and are applicable only to screens that allow bit-mapped graphics of those dimensions.

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
        3                GRAPHIC:        the screen when used for graphics
        4              <volume name>:    the System disk
        5              <volume name>:    the alternate disk
        6                PRINTER:        the line printer
        7                REMIN:          serial line input
        8                REMOUT:         serial line output
      9-12             <volume name>:    additional disk drives
=====================================================================
```

TABLE 111.1

Note that device #3 (GRAPHIC:) only applies to users of Terak machines.

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.

lll.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.

⟨file specification⟩

volume ID → string → (=) / (?) → string →

[→ positive integer / (*) →]
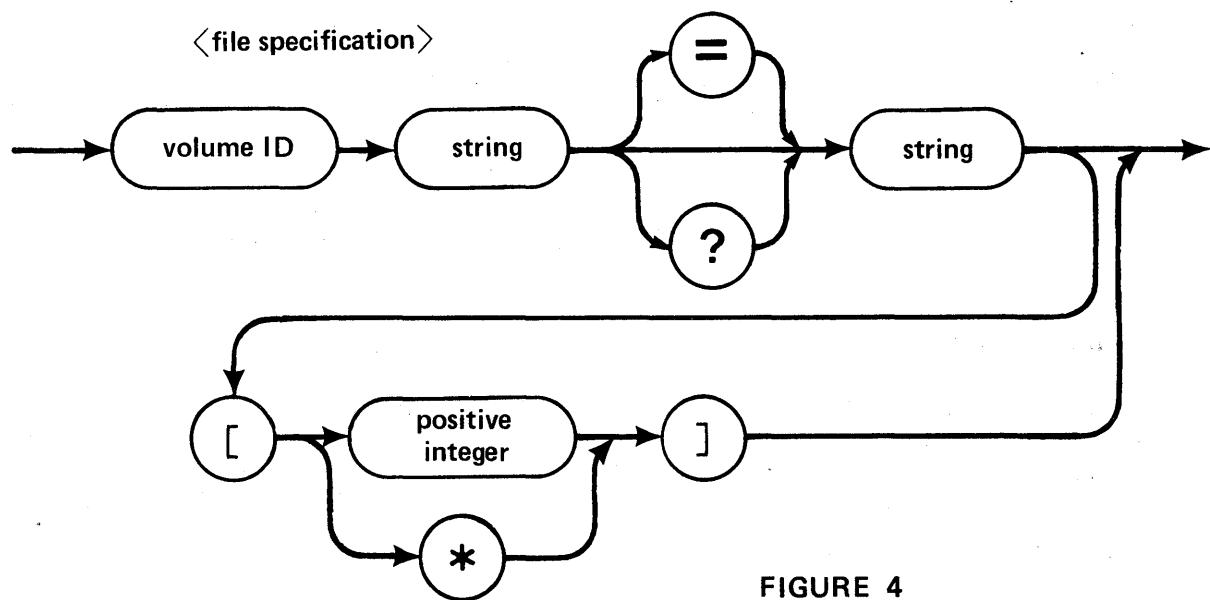
FIGURE 4

Volume i.d. syntax can be expanded thusly:
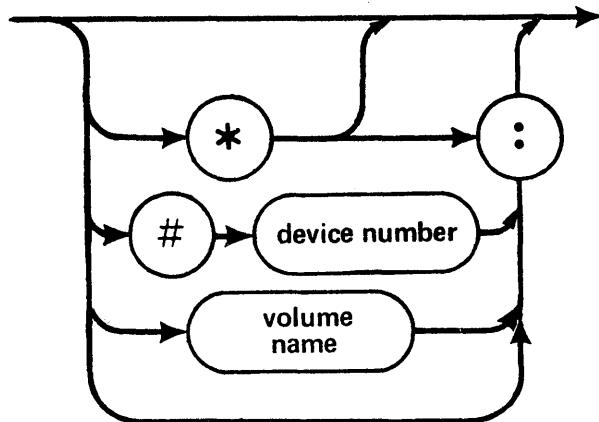
FIGURE 5

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 lll.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 <u>not</u> 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.4.3.   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, Q(uit [A ]`

Typing '?' in response to this prompt displays more Filer commands:

`Filer: B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(ols, X(amine, Z(ero`

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.   <u>Always</u> 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.

lll.6.2 Names of Files

lll.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 lll.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.

lll.6.2.2 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

lll.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

lll.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 494 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).

lll.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:

D(ate

lll.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.

lll.6.3.4   E(xtended list

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

All files <u>and unused areas</u> 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 lll.6.3.8, and Section lll.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, 356 unused, 200
in largest
```

G(et

lll.6.3.5   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, <u>regardless</u> 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.

lll.6.3.6   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. lt 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.

lf 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. lt 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.

lf 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. lf you do, the information on your disk may be irretrievably garbled.

EXAMPLE:

> Prompt: Crunch what vol?

> Response: MYDlSK:

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

> Prompt: From end of disk, block 494 ? (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.

lll.6.3.7   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, 350 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, 432 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, 432 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, 432 unused,
200 in largest

lll.6.3.8   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 lll.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.

lll.6.3.9   N(ew

Clears the workfile and creates a blank, unnamed workfile, which remains unnamed until it is saved.

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 <u>does</u> create a new workfile.

A successful N(ew returns the message:

> Workfile cleared

111.4.3.10   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, <u>any</u> 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 "*".

lll.6.3.11   Q(uit

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

lll.6.3.12   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 (inadvertantly) 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.)

lll.6.3.13   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

T(ransfer

lll.6.3.14   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 lll.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> (ASCll 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 Installation Guide 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.

111.6.3.15   V(olumes

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

A typical display might be:

```
Volumes on-line:
   1    CONSOLE:
   2    SYSTERM:
   4 #  MYD1SK:
   6    PRINTER:
   7    REMIN:
   8    REMOUT:
   9 #  BIG:
Root vol is  - MYD1SK:
Prefix is    - MYD1SK:
```

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

111.6.3.16   W(hat

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

EXAMPLE:

       Workfile is DOC1:STUFF

lll.6.3.17   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.

lll.6.3.18   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 494 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.

lll.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, 449 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 files<listed/in-dir>, 49 blocks used, 445 unused, 369 in largest

One further note: in order to eX(ecute a codefile, it is necessary that the codefile was <u>created</u> with a .CODE suffix; the codefile's name may later be changed. If the user has lost a codefile that does <u>not</u> 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 X.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 X. The following paragraph outlines the use of PATCH in this particular context; the reader should also refer to Section X.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..10.

Examining other blocks of a disk is a routine use of PATCH, described fully in Section X.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 X.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 X.7

86

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).

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.

Ken Bowles´ Beginner´s Guide is another good introduction to the Screen Oriented Editor, and we recommend that you use it.

Before the Screen Oriented Editor can be used, the System must first have been configured for a particular terminal.   The details of this are given in the Installation Guide, which you should refer to.

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(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap [E.6]

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 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. On terminals which do not have arrow keys, the System will have to be set up with a set of control keys to act as cursor keys. To configure the System for use with a particular terminal, please refer to the Installation Guide.

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 <u>before</u> 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 <u>not</u> 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 lV.4 (Screen after typing 'SMART')
```
=====================================================================
BEGIN WRITE('TOO SMART                                       WISE ');
=====================================================================
```

Figure lV.5 (Screen after <etx>)
```
=====================================================================
BEGIN
   WRITE('TOO SMARTWISE ');
=====================================================================
```

Figure lV.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 l(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.

95

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 1.

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 -- either a vector pad, or a set of control characters that you have chosen yourself (see the sections concerning SETUP in the Installation Guide). 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 <u>not</u> 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 <u>any</u> 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 overridden 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.

A(djust

IV.3.1 A(djust

On the promptline: A(djst.

Repeat factors are allowed.

A(djust displays the following prompt:

>Adjust: L(just R(just C(enter <left,right,up,down-arrows> {<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(py.

Repeat factors not allowed.

C(opy displays the following promptline:

>C(opy: B(uffer F(ile   <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(lete.

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 1V.8
```
===================================================================
PROGRAM STRING2;
BEGIN
  WRITE('TOO WISE ');
  WRITELN('TO BE.')
END.
===================================================================
```

Figure 1V.9
```
===================================================================
PROGRAM STRING2:
BEGIN
END.
===================================================================
```

In Figure 1V.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 1V.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 <u>after</u> 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), <u>before</u> 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.

108

IV.3.5   I(nsert

On the promptline: I(nsrt.

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.

J(ump

IV.3.6 J(ump

On the promptline: J(mp.

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

Not on the promptline; type ʹKʹ for K(olumn.

Repeat factors are not allowed.

K(olumn displays the following prompt:

>K(olumn: <vector keys> {<enter> to leave}

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).   <enter> (that is, <etx>) must be used to leave K(olumn; <esc> will not work.

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.

M(argin

IV.3.8   M(argin

Not on the promptline; type 'M' to use M(argin (which is also called M(unch).

Repeat factors not allowed.

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.

===================================================================
```

114

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.

Warning: 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

Not on the promptline; type 'P' to use 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 1.2.2.3, 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(place.

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

Not on the promptline; type 'S' to use 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:

Name of 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 ten 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.
0) name1
1) name2
.   ...
.   ...
9)name10
==

... appears, it is necessary to eliminate one marker in order to replace it. Choose a number 0 thru 9, type that number, and that space will now be available for use in setting the desired 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} <etx> or <sp> to leave
     A(uto indent   True
     F(illing        False
     L(eft margin   1
     R(ight margin  80
     P(ara margin   6
     C(ommand ch     ^
     T(oken def     True

     7436 bytes used,   12020 available

     Patterns:
       <target>= 'xyz', <subst>=   'abc'

     Date Created: 4-13-55    Last Used: 12-28-78


======================================================================
```

(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(nsertx. 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.

T(oken def:

This option affects F(ind and R(eplace. 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.

V(erify

IV.3.13   V(erify

Not on the promptline; type 'V' for 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

Not on the promptline; type 'X' for eX(change.

Repeat factors not allowed.

On entering eX(change the following promptline appears:

>eXchange: TEXT {<bs> a char} [<esc> escapes; <etx> accepts]

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   Figure IV.24
====================   ====================
WRITE('TOO wISE ');   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 may be specified with SETUP (see the Installation Guide).

Z(ap

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 read in'. 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.
RUBOUT \<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.
CTRL H \<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.

CTRL O   Switches you to the optional character
set (i.e., bit 7 is turned on). This <u>only</u>
works on the TERAK 8510A. CTRL O toggles
between the character sets. NOTE: You
may find while in YALOE that weird
characters are showing up on the terminal
instead of normal ones. Perhaps you
accidentally typed CTRL O. To get back,
just type CTRL O again.

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 1) 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 <u>only</u> 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 lll.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 underlined, 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 /Dl).

139

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 UCSD PASCAL LANGUAGE

VI.1   The UCSD Pascal Implementation

This section is a summary and reference which describes the areas in which UCSD Pascal differs from "standard" Pascal, or specifies details of topics (such as packing) which are normally implementation-dependent.   For a full description of the Pascal language, the user is referred to other appropriate references.   The standard Pascal referred to in this section is defined by the PASCAL User Manual and Report (2nd edition)   by Kathleen Jensen and Niklaus Wirth (New York: Springer-Verlag, 1975).

The reader may already be aware that there is an active effort underway to produce a formal international (as well as American) standard for Pascal.   This effort is co-ordinated internationally by the ISO (International Standards Organization) and domestically by ANSI (the American National Standards Institute). As the details of this standard solidify, and its use becomes more widespread, it will replace Jensen and Wirth as the reference for Pascal, and become the basis for comparisons with UCSD Pascal.   UCSD Pascal will continue to evolve in the direction of a commonly adopted standard, and will eventually comply with it.

It is recommended that the reader read all of Section VI.1.1, then use the information gained there to refer to the relevant parts of Sections VI.1.2 and VI.2. Section VI.1.2 covers variations from the standard, including differences in the implementation of standard procedures and functions -- special attention should be paid to these.   Section VI.2 lists, in alphabetical order, the various intrinsic functions which are not part of the standard language, but which have been included in this implementation.   Some of these are esoteric, but most are generally useful.

VI.1.1   An Introduction to the UCSD Implementation

There are five general areas in which the UCSD implementation differs from other implementations of Pascal:

1) String Handling: The type STRING has been added to the language, along with a number of intrinsic functions for manipulating strings.

2) I/O Intrinsics: A number of intrinsics have been added to facilitate handling of files and peripheral devices. The standard Pascal I/O intrinsics have been slightly modified to make them more useful in an interactive environment.

3) Separate Compilation and Memory Management: The language has been extended by the addition of SEGMENT routines, which facilitate swapping of program code at execution time, and UNITs, which allow separate compilation of Pascal routines and data structures.

4) Concurrency: Some syntax extensions have been made, and a few intrinsics added, to support the use of concurrent processes.

5) Miscellaneous: There are a number of small deviations and extensions to Pascal syntax, and limitations imposed by the microprocessor environment.

The sections below on strings, memory management, and concurrency (VI.1.2.1, VI.1.2.3, VI.1.2.4, and Chapter IX), need only be read by users interested in those particular features.   It is recommended that you do read the sections on I/O intrinsics and on miscellaneous deviations (VI.1.2.2 and VI.1.2.5), because these sections detail ways in which the UCSD use of Pascal may differ from a version you are familiar with.   Section VI.1.2 also points out the various UCSD intrinsics that are relevant to each particular area; non-standard intrinsics are described in alphabetical order in Section VI.2.

VI.1.2   Differences between UCSD Pascal and Standard Pascal

VI.1.2.1   Strings

UCSD Pascal has an additional predeclared type STRING.   A variable or constant of type STRING is a sequence of characters.   In a string variable, the length of the sequence can vary dynamically during the execution of a program.

A STRING variable has a maximum length (also known as the static length).   The default maximum length of a STRING variable is 80 characters.   This can be overridden in the declaration of a STRING variable by following the predeclared type identifier STRING with the desired maximum length enclosed in square brackets ([ ]).

Strings can be manipulated by either standard Pascal syntax, or by the special string-handling intrinsics in UCSD Pascal.   The UCSD intrinsic function LENGTH may be used to learn the dynamic length of a string.

Examples of STRING declarations:

```
        TITLE: STRING;   (* defaults to a maximum length *)
                         (* ... of 80 characters         *)

        NAME: STRING[20];   (* defines the STRING with a *)
                            (* maximum of 20 characters  *)
```

The maximum length a STRING variable may have is 255 characters.   The empty string (LENGTH = 0, represented by '') is allowed.

Values may be assigned to STRINGs using assignment statements, UCSD STRING intrinsics, or READ (READLN) statements.

Examples:

```
        TITLE:='     THIS IS A TITLE      ';
... or ...
        READLN(TITLE);
... or ...
        NAME:= COPY(TITLE,1,20);
```

The individual characters within a STRING are indexed as a PACKED ARRAY OF CHAR would be indexed, from 1 to the LENGTH of the STRING.

For example:

>   LETTERS[1]:= 'A';
>
>   LETTERS[LENGTH(LETTERS)]:= 'Z';

A variable of type STRING may not be indexed beyond its current dynamic LENGTH when range-checking is turned on. (Beware of strings of length zero!) The following sequence will result in an 'Invalid Index' runtime error:

>   TITLE:= '1234';
>   TITLE[5]:= '5';

Variables of type STRING are compatible for assignment and comparison with any other string constant or variable regardless of either static or dynamic length. A string may also be compared with a PACKED ARRAY OF CHAR. String comparisons return a result based on lexicographical ordering (i.e., lower case > upper case).

Example:

```
PROGRAM  COMPARESTRINGS;
VAR S:  STRING;
    T:  STRING[40];

BEGIN
  S:= 'SOMETHING';
  T:= 'SOMETHING BIGGER';
  IF S = T THEN
    WRITELN('Strings do not work very well')
  ELSE
    IF S > T THEN
      WRITELN(S,' is greater than ',T)
    ELSE
      IF S < T THEN
        WRITELN(S,' is less than ',T);
  IF S = 'SOMETHING' THEN
    WRITELN(S,' equals  ',S);
  IF S > 'SAMETHING' THEN
    WRITELN(S,' is greater than SAMETHING');
  IF S = 'SOMETHING                ' THEN
    WRITELN('BLANKS DON''T COUNT')
  ELSE
    WRITELN('BLANKS APPEAR TO MAKE A DIFFERENCE');
  S:='XXX';
  T:='ABCDEF';
  IF S > T THEN
    WRITELN(S,' is greater than ',T)
  ELSE
    WRITELN(S,' is less than ',T);
END.
```

The above program produces the following output:.

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING equals  SOMETHING
SOMETHING is greater than SAMETHING
BLANKS APPEAR TO MAKE A DIFFERENCE
XXX is greater than ABCDEF
```

147

One of the most common uses of STRING variables in UCSD Pascal is reading file names from the CONSOLE: device:

```
PROGRAM LISTER;
VAR BUFFER: PACKED ARRAY[0..511] OF CHAR;
    FILENAME: STRING;
    F: FILE;

BEGIN
  WRITE('Enter filename of the file to be listed -->');
  READLN(FILENAME);
  RESET(F,FILENAME);
  WHILE NOT EOF(F) DO
    BEGIN
       . . .
       . . .
       . . .
    END;
END.
```

When a STRING is to be read by the Pascal intrinsics READ or READLN, characters are read into that STRING variable one at a time, up to but not including the EOLN (<return>). Thus, only one STRING can be read per line of the input file.

For example, the single statement READLN(S1,S2) is equivalent to the two statement sequence READ(S1); READLN(S2). In both cases the STRING variable S2 will be assigned the empty string. (READ and READLN are described in Section VI.1.2.2.4.)

The string-handling intrinsics are: CONCAT, DELETE, INSERT, LENGTH, and POS. You should refer to the descriptions of these intrinsics in Section VI.2 below.

VI.1.2.2   I/O Intrinsics

In addition to the divergences from standard Pascal described in this section, the reader should also refer to the I/O intrinsics which have been added to the UCSD implementation.   These are BLOCKREAD, BLOCKWRITE, CLOSE, IORESULT, UNITBUSY, UNITCLEAR, UNITREAD, UNITSTATUS, UNITWAIT, and UNITWRITE. They are all described in Section VI.2 below.

EOF

VI.1.2.2.1   End of File - EOF

To set EOF TRUE for a textfile being entered from the CONSOLE:, the user must type the EOF character.   The EOF character is often called <etx> and often set to control-C.   The utility SETUP may be used to find out what character has been defined as <etx> in your System; it may also be used to change that setting. Refer to the section on SETUP in the Installation Guide.

If a file F is closed, EOF(F) will return the value TRUE.   For a TEXT file, EOF(F) = TRUE implies that EOLN(F) is also TRUE.   After a RESET(F), EOF(F) is FALSE.   If EOF(F) becomes TRUE during a GET(F) or a READ(F,...) the data obtained thereby is not valid.

When a user program starts execution, the system performs a RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD.   The predeclared file KEYBOARD is described in Section VI.1.2.2.4.

EOF and EOLN refer to the file INPUT unless the name of another file is given as their first parameter (this corresponds to Jensen and Wirth).

VI.1.2.2.2   End of Line -- EOLN

EOLN(F) is defined only if the contents of F are of type CHAR. EOLN(F) becomes TRUE only after reading a <return> from file F. As with EOF, if the first parameter is not the name of a file, EOLN refers to the standard file INPUT.

The following example is meant to show the importance of typing a <return> at the proper time:

```
PROGRAM ADDLINES;
VAR K,SUM:INTEGER;

BEGIN
   WHILE NOT EOF(INPUT) DO
      BEGIN
         SUM:=0;
         READ(INPUT,K);
         WHILE NOT EOLN(INPUT) DO
            BEGIN
               SUM:=SUM+K;
               READ(INPUT,K);
            END;
         WRITELN(OUTPUT);
         WRITELN(OUTPUT,'THE SUM FOR THIS LINE IS ', SUM);
      END;
END.
```

In order for EOLN(F) to be TRUE in the above program, <return> must be typed immediately after the last digit of the last integer on that line. If instead a space is typed followed by <return>, EOLN will remain FALSE and another READ will take place.

Vl.1.2.2.3   Files

Vl.1.2.2.3.1   Interactive Files

Files of type INTERACTIVE are composed of characters, just like files of type TEXT.   INTERACTIVE files differ from TEXT files in their behavior when they are used by the intrinsics READ, READLN, and RESET.   This behavior is intuitive in an interactive environment.

Typed files that are not INTERACTIVE behave exactly as described in Jensen and Wirth.

The standard predeclared files INPUT and OUTPUT are defined to be of type INTERACTIVE.   The file KEYBOARD (which is predeclared in UCSD Pascal) is also INTERACTIVE.

INPUT defaults to CONSOLE:.   The statement READ(INPUT,CH) where CH is a character variable, will echo the character typed from CONSOLE: back to CONSOLE:.

WRITE statements to OUTPUT will, by default, cause the output to appear on CONSOLE:.

KEYBOARD is the non-echoing equivalent to INPUT.   For example, the two statements:

```
READ(KEYBOARD,CH);
WRITE(OUTPUT,CH);
```

... are equivalent to the single statement READ(INPUT,CH).

For a way to "redirect" the standard files INPUT and OUTPUT, see Section 11.3.

Given the following declarations:

```
VAR CH:CHAR;
      F: TEXT;   (* TYPE TEXT = FILE OF CHAR *)
```

... the statement READ(F,CH) is defined by Jensen and Wirth to be equivalent to the two-statement sequence:

```
CH:=F^;     {    J & W     }
GET(F);     {    method    }
```

In other words, the standard definition of READ requires that opening a file must load the "window variable" F^ with the first character of the file.   In an

interactive programming environment, it is not convenient to require a user to type in the first character of the input file at the time the file is opened. If this were the case, every program using files would "hang" until a character was typed, whether or not the program performed any input operations at all.

In order to overcome this problem, READ(F,CH) on an INTERACTIVE file is the reverse of the sequence specified by the standard definition for files of type TEXT:, i.e.,

```
GET(F);        {  UCSD Pascal }
CH:=F^;        {     method    }
```

This difference affects the way in which EOLN must be used when reading from a textfile of type INTERACTIVE. As described above, EOLN becomes TRUE only after reading the end of line character (<return>). When a <return> is read, EOLN is TRUE, and the character returned as a result of the READ will be a blank.

On a standard file, RESET(F) performs an immediate GET(F). This does not take place if the file is INTERACTIVE. Thus, on an INTERACTIVE file, the equivalent of a standard RESET would be the two-statement sequence:

```
RESET(F);      {  makes INTERACTIVE }
GET(F);        {  look like TEXT    }
```

Refer to Section VI.1.2.2.4 on READ and READLN, and Section VI.1.2.2.5 on RESET for more details.

VI.1.2.2.3.2   Untyped Files

UCSD Pascal allows files to be declared without a type.

An untyped file F can be thought of as a file <u>without</u> a window variable F^. With such a file, all I/O must be performed using the functions BLOCKREAD and BLOCKWRITE.   BLOCKREAD and BLOCKWRITE are UCSD intrinsics (refer to Section VI.2 below).   Any number of blocks can be transferred using either BLOCKREAD or BLOCKWRITE.   The functions return the number of blocks transferred.

Example:

```
PROGRAM FILEDEMO;

VAR
   BLOCKNUMBER,BLOCKSTRANSFERRED:INTEGER;
   BADIO: BOOLEAN;
   G,F: FILE;
   BUFFER:   PACKED ARRAY[0..511] OF CHAR;

(* This program reads a diskfile called 'SOURCE.DATA' and
   copies the file into another diskfile called 'DESTINATION'
   using untyped files and the intrinsics BLOCKREAD and
   BLOCKWRITE *)

BEGIN
   BADIO:=FALSE;
   RESET(G,'SOURCE.DATA');
   REWRITE(F,'DESTINATION');
   BLOCKNUMBER:=0;
   (*$1-*)    {this turns off I/O checking}
   BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
   WHILE (NOT EOF(G)) AND (IORESULT=0) AND (NOT BADIO) AND
         (BLOCKSTRANSFERRED=1) DO
      BEGIN
         BLOCKSTRANSFERRED:=BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
         BADIO:=((BLOCKSTRANSFERRED<1) OR (IORESULT<>0));
         BLOCKNUMBER:=BLOCKNUMBER+1;
         BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
      END;
   CLOSE(F,LOCK);
END.
```

VI.1.2.2.3.3   Random Access of Files

Files may be randomly accessed by use of the UCSD intrinsic SEEK. SEEK expects two parameters: the file identifier, and an integer specifying the record number to which the window should be moved. The first record of a structured file is record number 0.

Attempts to PUT records beyond the physical end of file will set EOF to the value TRUE. (The physical end of file is the point where the next record in the file would overwrite another file on the disk.) SEEK always sets EOF and EOLN to FALSE. The subsequent GET or PUT will set these conditions as is appropriate. SEEK is described in Section VI.2.

The following sample program demonstrates the use of SEEK to randomly access and update records in a file:

```
      PROGRAM RANDOMACCESS;
      VAR
        RECNUMBER: INTEGER;
        CH: CHAR;
        DISK: FILE OF RECORD
                        NAME: STRING[20];
                        DAY,MONTH,YEAR: INTEGER;
                        ADDRESS: PACKED ARRAY[0..49] OF CHAR;
                        ALIVE: BOOLEAN
                     END;
      BEGIN
        RESET(DISK,'RECORDS.DATA');

        WHILE NOT EOF(INPUT) DO
        BEGIN
          WRITE(OUTPUT,'Enter record number -->');
          READ(INPUT,RECNUMBER);

          SEEK(DISK,RECNUMBER);
          GET(DISK);

          WITH DISK^ DO
            BEGIN
              IF NOT EOF(DISK) THEN
                WRITELN(OUTPUT,NAME,DAY,MONTH,YEAR,ADDRESS)
              ELSE
                WRITELN('New Record');
              WRITE(OUTPUT,'Enter correct name -->');
              READLN(INPUT,NAME);
              ...
              ...
            END;

          (* Must point the window back to the record
             since GET(DISK) advances the window to
             the next record after loading DISK^       *)

          SEEK(DISK,RECNUMBER);
          PUT(DISK);
        END;
      END.
```

VI.1.2.2.3.4 Files as Elements of Records or Arrays

UCSD Pascal does not allow files to be declared inside structured variables (i.e., arrays or records). One consequence of this is that file variables cannot be stored on the heap. This restriction is imposed so that the Compiler can easily emit hidden code to open and close an internal file at the proper limits of its scope.

READ and READLN

Users' Manual
UCSD Pascal

VI.1.2.2.4   READ and READLN

Strings are read character-by-character, until terminated by a <return>.

When integers are read, leading blanks and end-of-lines are flushed until a non-blank character is read.   An integer is terminated by a space (' '), a character that is not a digit,   or a <return>.   Before it has been completely read, it may be corrected by <backspace>'ing over it and re-typing.

Reals are read in the same way as integers.

Booleans may not be read.   Neither may any structured type.

The behavior of READ and READLN conforms to the definition in Jensen and Wirth, except when handling files that are INTERACTIVE.   The standard file INPUT is defined to be INTERACTIVE.   The action of READ on an INTERACTIVE file is described in Section VI.1.2.2.3.1 and illustrated below.

In the following example, the left fragment is taken from Jensen and Wirth; only the RESET and REWRITE statements have been altered.   The program on the left will correctly copy the textfile represented by the file X to the file Y. The program fragment on the right performs a similiar task, except that the source file being copied is INTERACTIVE, thus forcing a slight change in the program in order to produce the desired result.

```
PROGRAM JANDW;                          PROGRAM UCSDVERSION;
VAR X,Y:TEXT;                           VAR X,Y:INTERACTIVE;
     CH: CHAR;                               CH:CHAR;
BEGIN                                   BEGIN
   RESET(X,'SOURCE.TEXT');                 RESET(X,'CONSOLE:');
   REWRITE(Y,'SOMETHING.TEXT');            REWRITE(Y,'SOMETHING.TEXT');
                                           READ(X,CH);
   WHILE NOT EOF(X) DO                     WHILE NOT EOF(X) DO
     BEGIN                                   BEGIN
        WHILE NOT EOLN(X) DO                    WHILE NOT EOLN(X) DO
          BEGIN                                   BEGIN
             READ(X,CH);                             WRITE(Y,CH);
             WRITE(Y,CH);                            READ(X,CH);
          END;                                    END;
        READLN(X);                              READLN(X);
        WRITELN(Y);                             WRITELN(Y);
     END;                                    END;
   CLOSE(Y,LOCK);                          CLOSE(Y,LOCK);
END.                                    END.
```

Note that the textfiles X and Y in both programs had to be opened by using the UCSD extended form of the standard procedures RESET and REWRITE.

The CLOSE intrinsic (a UCSD intrinsic; see Section VI.2) was applied to the file Y in both versions of the program in order to make it a permanent file in the disk directory called 'SOMETHING.TEXT'. Likewise, the textfile X could have been a diskfile instead of coming from CONSOLE: in the right-hand version of the program.

RESET

VI.1.2.2.5   RESET

The standard procedure RESET(F) resets the file window to the beginning of the file F.   The next GET(F) or PUT(F) affects record number 0 of that file.   An immediate GET(F) is also performed within RESET (thus GETting the first record of the file), unless the file F is of type INTERACTIVE.

Thus, for INTERACTIVE files, the UCSD equivalent of the standard definition of RESET(F) is the two-statement sequence:

```
RESET(F);      {  makes INTERACTIVE  }
GET(F);        {  look like TEXT      }
```

(Except for this stipulation about INTERACTIVE files, the behavior of RESET conforms to Jensen and Wirth.)

The UCSD implementation also allows RESET to have a second parameter, which is the name of an existing disk file or device, contained in a string constant or string variable.   The disk file (or device) is referred to as an "external" file; the file that is a data object in the Pascal program is called an "internal" file.

For example:

```
RESET(F, 'ODD.TEXT')

RESET(F, FNAME)
```

... are statements that associate the internal (Pascal) file F with the external (disk) file 'ODD' or the disk file named in FNAME.

Trying to RESET a nonexistent external file, or an internal file that is already open, will cause an I/O error.   Trying to RESET a write-only device (such as PRINTER:) will cause an I/O error, since the device is not an input device, and the GET that RESET implicitly performs will attempt to read the device.

External files that are opened by a program with RESET or REWRITE may be closed with the UCSD intrinsic CLOSE.   See Section VI.2.

VI.1.2.2.6   REWRITE

The intrinsic REWRITE "clears" a file by setting F to the empty file, and EOF(F) to TRUE.   A call to REWRITE may also be used to open a new file.

In UCSD Pascal, the REWRITE intrinsic may be called with a second parameter. The second parameter is the name of a disk file (as in RESET), contained in a string constant or a string variable.

If the disk file is named, it may be either an existing file, or a new file. If it is new, a file of the appropriate type is created on disk.   If it already exists, REWRITE creates a temporary file which can either supplant the old file, be saved under a new name, or discarded -- see the CLOSE intrinsic (in Section VI.2).

If there is no second parameter, then REWRITE(F) is equivalent to REWRITE('F') (only for the first 8 characters).

Trying to REWRITE an already open internal file causes an I/O error.

Aside from the provision for binding an internal file to an external filename, REWRITE behaves as defined in Jensen and Wirth.

WRITE and WRITELN

VI.1.2.2.7 WRITE and WRITELN

WRITE and WRITELN can write values of type INTEGER, REAL, STRING, and PACKED ARRAY OF CHAR. Booleans, other types of arrays, and other structured types, cannot be output.

UCSD's WRITE and WRITELN can write an entire PACKED ARRAY OF CHAR in a single WRITE statement:

```
VAR BUFFER: PACKED ARRAY[0..10] OF CHAR;
BEGIN
   BUFFER:= 'HELLO THERE';
      (* contains exactly 11 characters *)
   WRITELN(OUTPUT, BUFFER);
END.
```

Field width specifications work for STRINGs as well.

Example:

```
PROGRAM WRITESTRINGS;
VAR S:STRING;

BEGIN
   S:='THE BIG BROWN FOX JUMPED...';
   WRITELN(S);
   WRITELN(S:30);
   WRITELN(S:10);
END.
```

... produces the following output:

```
THE BIG BROWN FOX JUMPED...
    THE BIG BROWN FOX JUMPED...
THE BIG BR
```

When a string variable or constant is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length of the string, leading blanks are inserted and written. If the field width is smaller than the dynamic length of the string, the excess characters are truncated on the right.

VI.1.2.3 Separate Compilation and Memory Management

Separate compilation, memory management, and the management of codefiles are topics discussed thoroughly in Chapter VIII: 'Segments, Units, and Linking'. This section will only show the syntax of particular extensions.

VI.1.2.3.1 Memory Allocation

The standard procedures DISPOSE and NEW are implemented. The MARK/RELEASE mechanism used in earlier versions of UCSD Pascal is still supported. In addition, the following UCSD intrinsics are provided as aids to memory management: MEMAVAIL, VARAVAIL, VARDISPOSE, VARNEW. These are described in Section VI.2. If you intend to make much use of direct control of memory resources, you should refer to the Internal Architecture Guide.

Important: if you NEW a record with a particular variant record, you must DISPOSE that record using the same variant -- otherwise, you risk damaging the heap and hence crashing the System. Similarly, it is crucial that MARKs and RELEASEs be properly paired: the contents of a MARKed pointer must not be altered until the matching call to RELEASE has been performed, and RELEASEs must only be performed on variables that are MARKed but not yet RELEASEd.

VI.1.2.3.2   Segment Routines

Routines (i.e., procedures, functions or processes) normally occupy the same code segment as the compilaton unit in which they appear.   A segment routine occupies a code segment of its own.   Code is swapped into memory a segment at a time, and the space a segment occupies in memory becomes available to other programs as soon as it is no longer in use.   Thus, declaring some routines (such as a program's initialization and termination routines, for instance) as segment routines may improve a program's utilization of main memory.

A routine is made a segment routine by preceding its declaration with the reserved word 'SEGMENT'.

Example:

```
    SEGMENT PROCEDURE ONE;
      BEGIN
        PRINT('SEGMENT NUMBER ONE');
      END;
```

More information about segment routines (in particular, some restrictions on the way they must be declared) appears in Chapter VIll.

VI.1.2.3.3 Units

UNITs are a mechanism for compiling Pascal routines and data structures separately from the main program. This is useful in preparing long programs that compile slowly, or in co-ordinating the efforts of several programmers by using common facilities. UNITs are described fully in Chapter VIII.

A UNIT may be compiled by itself, or it may be declared within a program source file.

A program or unit (the "client" or "host") may use another unit by naming it in a USES declaration. The USES clause immediately follows the program heading, or appears in a UNIT at the beginning of either the INTERFACE or IMPLEMENTATION sections.

Example:

```
PROGRAM SPAM;
   USES VIKINGS;
```

The UNIT itself consists of two main parts: an INTERFACE part which contains data declarations and procedure headings, and an IMPLEMENTATION part which contains more declarations and procedure bodies. The IMPLEMENTATION part is strictly private to the UNIT. The data structures and routines declared in the INTERFACE part of a UNIT may be used by a client as if they were declared in the client itself.

A UNIT may also contain an optional section of Pascal code, following the INTERFACE and IMPLEMENTATION parts. This is itself divided into two parts: initialization code, and termination code, separated by '***;'. The initialization code is executed <u>before</u> any host program code is executed, and the termination code is executed <u>after</u> the host program has completed execution.

The general outline of a unit is as follows:

```
UNIT EASE;

INTERFACE          {this is a UCSD reserved word}
    USES ...;      {optional}
    {declarations and procedure headings}

IMPLEMENTATION     {also a reserved word}
    USES ...;      {optional}
    {declarations and procedure code}

BEGIN
    {initialization code}
    ***;
    {termination code}
END  {of EASE};
```

VI.1.2.3.4   External Routines

A Pascal host may use a separately assembled routine.   The host must include a Pascal routine heading (with parameters, if there are any), and designate it EXTERNAL.

Examples:

FUNCTION FAST_AND_DIRTY (SPEED: INTEGER): BOOLEAN; EXTERNAL;
PROCEDURE WRITE_OUT; EXTERNAL;

Assembled routines that are meant to be used by a Pascal host must strictly adhere to Pascal calling conventions and System constraints on the use of resources such as memory and registers.   See Chapter VII for more details.   Before a host which uses external routines may be run, the routines must be bound to the host's code by using the Linker.   The Linker is described in Chapter VIII.

VI.1.2.4   Concurrent Processes

In UCSD Pascal, the user may declare a PROCESS.   A process declaration is similar to a procedure declaration, for example:

PROCESS GOLUX (VAR TODAL: REAL);

A process is a routine whose execution appears to proceed at the same time as (i.e., concurrently with) the main program.   Processes are initiated by the UCSD intrinsic START (see Section VI.2), for example:

START (GOLUX(5.417));

START has a few optional parameters which allow the user to specify the space allocation and the priority of a process.

The predeclared type SEMAPHORE allows concurrent processes to communicate with each other.   Semaphores are initialized by the UCSD intrinsic SEMINIT, and managed by the UCSD intrinsics SIGNAL and WAIT.   In addition, a semaphore may be associated with an external (i.e., hardware) interrupt by using the UCSD intrinsic ATTACH.

All of these intrinsics are described in Section VI.2, and the use of concurrent processes is discussed more fully in Chapter IX.

VI.1.2.5   Miscellaneous Differences

VI.1.2.5.1   CASE Statements

In UCSD Pascal, CASE statements "fall through" if there is no label equal to the case selector. The statement following the CASE statement is executed next.

For example, the following sample program will only output the line "THAT'S ALL FOLKS" since the case statement will "fall through" to the WRITELN statement following the case statement:

```
PROGRAM FALLTHROUGH;
VAR CH:CHAR;
BEGIN
  CH:='A';
  CASE CH OF
    'B': WRITELN(OUTPUT,'HI THERE');
    'C': WRITELN(OUTPUT,'THE CHARACTER IS A ''C''')
  END;
  WRITELN(OUTPUT,'THAT''S ALL FOLKS');
END.
```

Comments

Vl.1.2.5.2   Comments

The Compiler recognizes any text appearing between either the symbols '(*' and '*)' or the symbols '{' and '}' as a comment.   Text appearing between these symbols is ignored by the Compiler unless the first character of the comment is a dollar sign, in which case the comment is interpreted as a Compiler control comment.   These control comments are described in Section Vl.3.

If the beginning of the comment is delimited by the '(*' symbol, the end of the comment must be delimited by the matching '*)' symbol, rather than the '}' symbol.   When the comment begins with the '{' symbol, the comment continues until the matching '}' symbol appears.   This feature allows a user to "comment out" a section of a program which itself contains comments.   For example:

```
{      XCP := XCP + 1;   (* ADJUST FOR SPECIAL CASE... *)    }
```

The Compiler does <u>not</u> keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol.   The following text will result in a syntax error:

```
(*THIS IS A COMMENT   (*NESTED COMMENT*)   END OF FIRST COMMENT *)
                                            ^error here.
```

VI.1.2.5.3 Extended Comparisons

UCSD Pascal allows = and <> comparisons of any array or record structure.

GOTO and EXIT

Users' Manual
UCSD Pascal

VI.1.2.5.4   GOTO and EXIT Statements

A GOTO statement causes a "jump" in the flow of control of a program.   The next statement executed is the statement with the label named in the GOTO statement, and execution proceeds from that point.   The label and the GOTO statement <u>must</u> be within the same routine (or within the same main program block).   This is a more restricted form of the GOTO statement than that in the standard language.

EXIT is a UCSD extension which accepts as its single parameter the identifier of a routine to be exited, the identifier of a program, or the reserved word PROGRAM. EXIT causes the routine or program it names to be terminated immediately (but cleanly).

The use of an EXIT statement to exit a FUNCTION can result in the FUNCTION returning undefined values: this happens if no assignment has been made to the FUNCTION identifier prior to the execution of the EXIT statement.

An example of the use of the EXIT statement:

```
PROGRAM EXITDEMO;
VAR T: STRING;
    CN: INTEGER;

   PROCEDURE Q; FORWARD;

   PROCEDURE P;
   BEGIN
     READLN(T);
     WRITELN(T);
     IF T[1]='#' THEN EXIT(Q);
     WRITELN('LEAVE P');
   END;

   PROCEDURE Q;
   BEGIN
     P;
     WRITELN('LEAVE Q');
   END;

   PROCEDURE R;
   BEGIN
     IF CN <= 10 THEN Q;
     WRITELN('LEAVE R');
   END;

BEGIN
  CN:=0;
  WHILE NOT EOF DO
     BEGIN
        CN:=CN+1;
        R;
        WRITELN;
     END;
END.
```

If the above program is supplied the following input

> THIS IS THE FIRST STRING
> #
> LAST STRING

... the following output results:

> THIS IS THE FIRST STRING
> LEAVE P
> LEAVE Q
> LEAVE R
>
> #
> LEAVE R
>
> LAST STRING
> LEAVE P
> LEAVE Q
> LEAVE R

The EXIT(Q) statement causes the PROCEDURE P to be terminated followed by the PROCEDURE Q. Processing continues following the call to Q inside PROCEDURE R. Thus, the only line of output following '#' is 'LEAVE R' at the end of PROCEDURE R. In the two cases where the EXIT(Q) statement is not executed, processing proceeds normally through the terminations of procedures P and Q.

If the procedure identifier passed to EXIT is a recursive procedure, the most recent invocation of that procedure will be exited. If, in the above example, one or both of the procedures P and Q declared and opened some local files, an implicit CLOSE(F) is done when the EXIT(Q) statement is executed, as if the procedures P and Q terminated normally.

The EXIT statement may also be used to exit a Pascal program by EXIT(PROGRAM) or EXIT(programname).

The addition of the EXIT statement to UCSD Pascal was inspired by the occasional need for a straightforward means to abort a complicated and possibly deeply nested series of procedure calls upon encountering an error. Excessive use of the EXIT statement is discouraged.

VI.1.2.5.5   Long Integers

With the predeclared type INTEGER, a length attribute may optionally be included. INTEGERs declared in this way are called LONG INTEGERs.   The LONG INTEGER is intended for business, scientific, or other applications which need extended number lengths with complete accuracy.

This extension supports the four basic standard INTEGER arithmetic operations (addition, subtraction, multiplication, and division), as well as routines that facilitate conversion to strings and standard INTEGERs.   Input/Output, in-line declaration of constants, and inclusion in structured types, are all fully supported and are analogous to standard INTEGERs.

LONG INTEGERs are declared using the standard identifier INTEGER followed by a length attribute in square brackets.   This length is an unsigned number no greater than 36.

The length attribute specifies the minimum number of decimal digits the LONG INTEGER must be able to represent.

Example:

        VAR X: INTEGER[8];

... is an integer with a minimum of eight digits.

Constants are defined in the normal manner:

        CONST RYDBERG = 10973731;

In the above example RYDBERG would be by default (because of its magnitude) a LONG INTEGER, and could be used anywhere a LONG INTEGER could be used.

In general, LONG INTEGERs may be used anywhere it is syntactically correct to use REALs.   However, care must be taken to ensure that sufficient words have been allocated by the declared length attribute for storage of the result of assignment or arithmetic expression statements.   INTEGER expessions are implicitly converted to LONG INTEGERs as required by the space demands of an operation or assignment.   The reverse is not true.

Examples:

```
VAR  1:  INTEGER;
     L:  INTEGER[N]; {where N is an integer constant
                           <= 36 }
     S:  REAL;

     1:= L;  {syntax error, see TRUNC(L) below}
     L:=-L;  {correct}
     L:= 1;  {always correct}
     L:= S;  {never accepted}
     S:= L;
```

Arithmetic operations which may be used in conjunction with LONG INTEGERs are: +, -, *, DIV, unary plus/minus. On assignment, the length of the LONG INTEGER is adjusted (during execution) to the declared length attribute of the destination variable. Overflow may result if the destination variable is not large enough to hold the source.

Overflow checking and storage allocation for long integers both depend on two aspects of the way long integers are implemented:

1) an integral number of words is always assigned to a long integer, and

2) the internal representation of the long integer is implementation-dependent. (Binary integers, packed BCD, and radix 100 formats have all been used to represent long integers on various p-System implementations.)

The Compiler determines how much memory to allocate a long integer, and assumes the worst case about the storage efficiency of the representation. Overflow errors occur when the number of words required to represent the source long integer is greater than the space available in the destination long integer -- this may vary from machine to machine.

The comparisons =, <, >, <=, >=, and <> may be used in expressions that contain both LONG INTEGERs and INTEGERs.

The function TRUNC(L), where L is a LONG INTEGER, converts L to an INTEGER (i.e. TRUNC accepts both LONG INTEGERs and REALs as arguments). Overflow will result if L is greater than MAXINT or less than -MAXINT.

The procedure STR(L,S) converts the INTEGER or LONG INTEGER L, into a string (complete with minus sign if needed), and places it in the STRING S. The following program fragment illustrates a suitable "dollar and cent" routine:

```
STR(L,S); INSERT('.',S,LENGTH(S)-1); WRITELN(S);
```

Pascal syntax requires that parameter types be specified by type identifiers. Therefore, an attempt use an 'INTEGER[<length>]' style declaration in a parameter list will result in a syntax error, which may be circumvented by declaring an appropriate type identifier. For example:

        TYPE LONG = INTEGER[18];
        PROCEDURE BIGNUMBER(BANKACCT: LONG);

V1.1.2.5.6   Packed Variables

V1.1.2.5.6.1   Packed Arrays

UCSD Pascal will perform packing of arrays and records if the ARRAY or RECORD declaration is preceded by the word PACKED. For example, consider the following declarations:

        A: ARRAY[0..9] OF CHAR;

        B: PACKED ARRAY[0..9] OF CHAR;

The array A will occupy ten 16-bit words of memory, with each element of the array occupying 1 word. The PACKED ARRAY B, on the other hand, will occupy a total of only 5 words, since each 16-bit word contains two 8-bit characters. In this manner each element of the PACKED ARRAY B is 8 bits long.

PACKED ARRAYs need not be restricted to arrays of type CHAR.
For example:

        C: PACKED ARRAY[0..1] OF 0..3;

        D: PACKED ARRAY[1..9] OF SET OF 0..15;

        D2: PACKED ARRAY[0..239,0..319] OF BOOLEAN;

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range 0..3. Therefore, C occupies only one 16-bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9-word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of D2 in the above example, occupies only one bit.

The following two declarations are not equivalent:

        E: PACKED ARRAY[0..9] OF ARRAY[0..3] OF CHAR;

        F: PACKED ARRAY[0..9,0..3] OF CHAR;

The second occurrence of the reserved word ARRAY in the declaration of E causes the packing option in the Compiler to be turned off, so that E becomes an unpacked array of 40 words. On the other hand, the PACKED ARRAY F occupies 20 total words, because the word ARRAY occurs only once in the declaration. If E had been declared as:

        E: PACKED ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;

... or as ...

      E: ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;

... F and E would have had identical configurations.

PACKED only has true significance before the last appearance of the word ARRAY in a declaration of a PACKED ARRAY. When in doubt, a good rule of thumb is to place the word PACKED before <u>every</u> appearance of the word ARRAY, to ensure that the resulting array will be PACKED.

The resulting array will <u>only</u> be packed if the final type of the array is a scalar, subrange, or set which can be represented by 8 bits or less. The following declaration will result in no packing whatsoever, because the final type of the array cannot be represented in a field of 8 bits:

      G: PACKED ARRAY[0..3] OF 0..1000;

... G will be an array which occupies 4 16-bit words.

Packing <u>never</u> occurs across word boundaries. This means that if the type of the element to be packed requires a number of bits that do not divide evenly into 16, there will be some unused bits at the high order end of each of the words which comprise the array.

Note that for the purposes of assignment and comparison, a string constant is compatible with a <u>PACKED</u> ARRAY OF CHAR (but not an unpacked ARRAY OF CHAR). In a similar fashion, <u>no</u> packed array or record may be assigned to or compared with an unpacked version (of the corresponding type).

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the UCSD intrinsics FILLCHAR and SIZEOF:

```
PROGRAM FILLFAST;
VAR A: PACKED ARRAY[0..10] OF CHAR;
BEGIN
  FILLCHAR(A[0],SIZEOF(A),' ');
END.
```

The above sample program fills the entire PACKED ARRAY A with blank. Refer to the descriptions of FILLCHAR and SIZEOF in Section VI.2.

Vl.1.2.5.6.2   Packed Records

The following RECORD declaration declares a RECORD with 4 fields.   The entire RECORD occupies one 16-bit word, because it is declared as a PACKED RECORD.

```
VAR R:  PACKED RECORD
            1,J,K:  0..31;
            B:  BOOLEAN
        END;
```

The variables 1, J, K each take up 5 bits in the word.   The Boolean variable B is allocated the 16'th bit of the same word.

In much the same manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS.   Again, slight differences in the way in which declarations are made will affect the degree of packing achieved.   For example, note that the following two declarations are not equivalent:

```
VAR A:                               VAR B:
   PACKED RECORD                        PACKED RECORD
      C: INTEGER;                          C: INTEGER;
      F:PACKED RECORD                      F:RECORD
            R:  CHAR;                          R:CHAR;
            K:  BOOLEAN                         K:BOOLEAN
         END;                               END;
      H:PACKED ARRAY[0..3] OF CHAR       H:PACKED ARRAY[0..3] OF CHAR
END;                                 END;
```

As with packed arrays, the word PACKED must appear with every occurrence of the reserved word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD.   In the above example, only RECORD A has all of its fields packed into one word.   In B, the F field is not packed and therefore occupies two 16-bit words.   It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary.   This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

When a record (whether packed or not) contains a case variant, the field is allocated enough space to contain the largest variant.   Consider the following example:

```
VAR K: PACKED RECORD
         B: BOOLEAN;
         CASE F: BOOLEAN OF
            TRUE:   (Z:INTEGER);
            FALSE: (M: PACKED ARRAY[0..3] OF CHAR)
         END
      END;
```

In the above example, the B and F fields are stored in two bits of the first 16-bit word of the record. The remaining 14 bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire PACKED RECORD will occupy 3 words.

VI.1.2.5.6.3   Restrictions on Packed Variables

No element of a PACKED ARRAY or field of a PACKED RECORD may be passed as a variable (call-by-reference) parameter to a routine.   Packed variables may, however, be passed as value parameters.   This conforms to Jensen and Wirth.

UCSD Pascal does not support the standard procedures PACK and UNPACK (defined in Jensen and Wirth).   If a type or variable is declared as packed, the packing and unpacking is implicit.

VI.1.2.5.7   Parametric Procedures and Functions

UCSD Pascal does not allow PROCEDURES or FUNCTIONS to be declared as formal parameters in the parameter list of a PROCEDURE or FUNCTION.

VI.1.2.5.8 Program Headings

Although the UCSD Pascal Compiler will permit a list of file parameters to be present following the program identifier, these parameters are ignored by the Compiler and will have no effect on the program being compiled. As a result the following two program headings are equivalent:

```
                PROGRAM DEMO(INPUT,OUTPUT);
... and ...
                PROGRAM DEMO;
```

With either of the above program headings, a user program will have three files predeclared and opened by the System. These are: INPUT, OUTPUT, and KEYBOARD and are defined to be of type INTERACTIVE. If the programmer wishes to declare any additional files, these file declarations must be declared together with the program's other VAR declarations.

VI.1.2.5.9   Sets

Sets are defined as in Jensen and Wirth.   Sets of subranges of integers are limited to the positive integers.   The ordinal value of the upper bound of a set declaration must be no greater than 4079.   Note that the ordinal value of the lower bound of the set declaration does not affect this limit.

Comparisons and operations on sets are allowed only between sets which are either of the same base type or subranges of the same underlying type.   For example, in the sample program below, the base type of the set S is the subrange type 0..49, while the base type of the set R is the subrange type 1..100.   The underlying type of both sets is the type INTEGER, which by the above definition of compatibility, implies that the comparisons and operations on the sets S and R in the following program are legal:

```
PROGRAM SETCOMPARE;
VAR  S:  SET OF 0..49;
     R:  SET OF 1..100;

BEGIN
  S:= [0,5,10,15,20,25,30,35,40,45];
  R:= [10,20,30,40,50,60,70,80,90];
  IF S = R THEN
    WRITELN('... oops ...')
  ELSE
    WRITELN('sets work');
  S := S + R;
END.
```

In the following example, the construct I = J is <u>not</u> legal, since the two sets are of two distinct underlying types.

```
PROGRAM ILLEGALSETS;
TYPE STUFF=(ZERO,ONE,TWO);
VAR I: SET OF STUFF;
    J: SET OF 0..2;

BEGIN
  I:= [ZERO];
  J:= [1,2];
  IF I = J THEN ...       <<<< error here
END.
```

Vl.1.2.5.10   Transcendental Functions

The arctangent function may be called by <u>either</u> ATAN or ARCTAN.

Since Pascal has a rather limited set of transcendentals, the following list of formulas may prove useful:

```
asin(x)   =   atan( x / sqrt( 1 - x*x ) )
acos(x)   =   pi/2 - asin(x)
acot(x)   =   pi/2 + atan(-x)

if  x >= 0,
    asec(x)   =   atan( sqrt( x*x - 1 ) )
else
    asec(x)   =   atan( sqrt( x*x - 1 ) ) - pi

acsc(x)   =   asec( x / sqrt( x*x - 1 ) )
```

VI.1.2.5.11   Size Limitations

The following is a list of size limitations that apply to the current implementation of UCSD Pascal (Version IV.0):

1. Local variables in a PROCEDURE or FUNCTION can occupy a   maximum of 16383 words of memory.

2. Maximum number of characters in a STRING variable is 255.

3. Maximum number of words allocated to a SET is 255.   Therefore, the maximum number of elements in a set is (255*16)=4080.

4. Maximum number of routines within a segment is 256.

VI.2   UCSD Intrinsics

This section contains descriptions of all the non-standard intrinsic procedures and functions that are part of the UCSD System and are callable from UCSD Pascal. The intrinsics are listed in alphabetical order.   While there are many new intrinsics, most users will find themselves using only a subset of the available routines, expanding that subset as they become more fluent with the System, or embark on more sophisticated (or larger!) projects.

Many indications as to the use of these intrinsics are given in Section VI.1.2.

Many of the intrinsics were created to provide necessary capabilities for System internals.   As such, they were designed for speed and knowledgeable use, and therefore provide little or nothing in the way of parameter checking.   Necessary range or validity checks are the responsibility of the user. Since some of these intrinsics do no checking for range validity, they may easily cause the System to die a horrible death.   Those intrinsics which are particularily dangerous are noted as such in their descriptions.

Required parameters are listed along with the function/procedure identifier. Optional parameters are in [square brackets].   The default values for optional parameters are described in the text.

ATTACH

VI.2.1 PROCEDURE ATTACH (SEM: SEMAPHORE; VECTOR: INTEGER)

Associates a semaphore with an external interrupt. When the hardware raises that interrupt it causes SEM to be SIGNAL'ed. See Chapter IX.

To de-attach an interrupt vector from a semaphore, the user may call ATTACH in the following way:

ATTACH (NIL, <vector number>)

... where <vector number> is the appropriate integer.

The exact value of a vector, and the hardware state it represents, is extremely hardware-dependent. These issues (for particular processors) are discussed in the Installation Guide.

VI.2.2   FUNCTION BLOCKREAD
(FILEID, ARRAY, BLOCKS, [RELBLOCK]) : INTEGER;

FILEID is an untyped file (see Section VI.1.2.2.3.2).

ARRAY can be any sort of array.   (It can actually be any sort of variable, since BLOCKREAD does no checking.)

BLOCKS is an integer.

BLOCKREAD reads BLOCKS number of blocks from FILEID into ARRAY, and returns the number of blocks actually read.

If the value returned does not equal BLOCKS, then either the end of file was encountered, or a read error occurred.

If the end of file is encountered, EOF will be TRUE.

If the optional parameter RELBLOCK is not present, records are read sequentially. Immediately after FILEID has been initialized with RESET or REWRITE, BLOCKREAD starts from block 0.   Successive BLOCKREADs continue to read sequential records, unless RELBLOCK is used, or FILEID re-initialized with RESET or REWRITE.

If RELBLOCK is present, it is the number of the block BLOCKREAD will start reading from, relative to block 0.

If the parameter ARRAY contains an index (e.g., BIG_TABLE[1024]), then BLOCKREAD will fill ARRAY starting with that element.   This does not work for packed arrays other than PACKED ARRAYs OF CHAR.   If ARRAY is in fact a record, it may contain a field specification -- filling will start from there (and the record must not be packed).

Some machines require that data be addressed on a word boundary, and so the user must be careful when dealing with sequences of type CHAR.

This is a **dangerous** intrinsic, as the bounds of ARRAY are not checked.   The user is responsible for seeing that no valuable memory is destroyed.

BLOCKWRITE

Users' Manual
UCSD Pascal

VI.2.3   FUNCTION BLOCKWRITE
              (FILEID, ARRAY, BLOCKS,[RELBLOCK]) : INTEGER;

FILEID is an untyped file (see Section VI.1.2.2.3.2).

ARRAY can be any sort of array. (Or anything else, as indicated in BLOCKREAD.)

BLOCKS is an integer.

BLOCKWRITE writes BLOCKS number of blocks from ARRAY into FILEID, and returns the number of blocks actually transferred.

If the value returned does not equal BLOCKS, the end of file was encountered, or a write error occurred.

If the end of file is encountered, EOF will be TRUE.

If the optional parameter RELBLOCK is not present, blocks will be transferred to FILEID sequentially. After FILEID is initialized with RESET or REWRITE, BLOCKWRITE starts with block 0. Successive calls to BLOCKWRITE continue writing sequentially, unless RELBLOCK is used, or FILEID is re-initialized with RESET or REWRITE.

If RELBLOCK is present, it is an integer indicating which block to start writing to, relative to block 0.

As with BLOCKREAD, a subscript on the parameter ARRAY causes the transfer to start with that element of ARRAY. Also as with BLOCKREAD, a record may have a field specification, and neither records nor arrays may be packed (except for PACKED ARRAYs of CHAR).

Some machines require that data be addressed on a word boundary, and so the user must be careful when dealing with sequences of type CHAR.

Also as with BLOCKREAD, this is a **dangerous** intrinsic, because of its lack of checking.

192

VI.2.4   CHAIN (EXEC_OPTIONS: STRING)

EXEC_OPTIONS is an execution option string as defined in Section 11.3.

A call to CHAIN causes the System to eX(ecute EXEC_OPTIONS <u>after</u> the calling program (the "chaining program") has terminated.   The effect is <u>that</u> of manually typing 'X' for eX(ecute, and then entering the characters in EXEC_OPTIONS. Neither the System promptline nor the eX(ecute prompt are displayed; the System goes on to immediately perform the actions indicated by EXEC_OPTIONS.

If a program (or sequence of programs) contains more than one call to CHAIN, the EXEC_OPTIONS are saved in a queue, and performed in a first-in-first-out fashion <u>before</u> control of the System is returned to the user.

A call to CHAIN with an empty string (e.g., "CHAIN('');") clears the queue.

An execution error or an error in an EXEC_OPTIONS string clears the queue, and returns the System to the user.   A call to EXCEPTION may also clear the queue; see the intrinsic EXCEPTION.

CHAIN is a procedure in the Operating System's COMMANDIO unit; to use it, a program or unit must declare 'USES COMMANDIO'.

CLOSE

V1.2.5   PROCEDURE CLOSE (FILEID [OPTION]);

FILEID is the name of an internal file.   Typically, it is a file that was opened with a previous RESET or REWRITE, and at that time associated with an external file in the p-System (usually a disk file).   See sections V1.1.2.2.5 and V1.1.2.2.6 on RESET and REWRITE.

OPTION need not be present.   If it is present, it may be: ', LOCK', or ', NORMAL', or ', PURGE', or ', CRUNCH'.   (Note the commas!)

If OPTION is not present or is NORMAL, then CLOSE simply sets the file state to closed.   If the file was opened using REWRITE and is a disk file, it is deleted from the directory.

If the file associated with FILEID is on a block-structured device (such as a disk) and was opened with a REWRITE, the LOCK option makes it permanent in the directory; otherwise, a NORMAL close is done.

If the file associated with FILEID is on a block-structured device, the PURGE option deletes it from the directory.   If the file associated with FILEID was a device and not a block-structured volume, the device goes off-line.   If no physical file or device was associated with FILEID, a NORMAL close is done.

The CRUNCH option LOCKs the file and truncates it at the point of last access. That is, the position of the last GET or PUT to the file is now the end of file.

All CLOSEs regardless of the option mark the file as closed and make the implicit variable FILEID^ undefined.   CLOSE on a CLOSE'd file does nothing.

VI.2.6   FUNCTION CONCAT (SOURCEs) : STRING

SOURCE is a string variable or constant or literal value. There may be any number of SOURCE strings separated by commas.

This function returns a string which is the concatenation of all the strings passed to it.

Example:

```
SHORTSTRING := 'THIS IS A STRING';
LONGSTRING  := 'THIS IS A VERY LONG STRING.';
LONGSTRING := CONCAT('START ',SHORTSTRING,'-',LONGSTRING);
WRITELN(LONGSTRING);
```

Prints:

    START THIS IS A STRING-THIS IS A VERY LONG STRING.

COPY

VI.2.7 FUNCTION COPY (SOURCE, INDEX, SIZE) : STRING

SOURCE is a string, INDEX and SIZE are integers.

This function returns a string containing SIZE characters copied from SOURCE starting at the INDEXth position in SOURCE.

Example:

```
TL := 'KEEP SOMETHING HERE';
KEPT := COPY(TL,POS('S',TL),9);
WRITELN(KEPT);
```

Will print:

    SOMETHING

VI.2.8   PROCEDURE DELETE (DESTINATION , INDEX , SIZE)

DESTINATION is a string, INDEX and SIZE are integers.

This procedure removes SIZE characters from DESTINATION starting at the INDEX specified.

Example (see the description of POS):

```
OVERSTUFFED := 'THIS STRING HAS FAR TOO MANY CHARACTERS IN IT.';
DELETE(OVERSTUFFED,POS('HAS',OVERSTUFFED)+3,8);
WRITELN(OVERSTUFFED);
```

Will print:

    THIS STRING HAS MANY CHARACTERS IN IT.

EXCEPTION

VI.2.9 PROCEDURE EXCEPTION (STOPCHAINING: BOOLEAN)

EXCEPTION turns off <u>all</u> redirection. If STOPCHAINING is TRUE, then the queue of EXEC_OPTIONS created by CHAIN is also cleared (see the intrinsic CHAIN).

Whenever an execution error occurs, an EXCEPTION(TRUE) call is made (to leave redirection on after an error would leave the System in an indeterminate state).

See Section II.3 for more information on redirection.

EXCEPTION is a procedure in the Operating System's COMMANDIO unit; to use it, a program or unit must declare 'USES COMMANDIO'.

VI.2.10   PROCEDURE FILLCHAR
         (DESTINATION, LENGTH, CHARACTER);

DESTINATION may be any sort of array. It may be subscripted. (It may also be any other sort of variable, at the user's risk. If it is a record, it may have a field specification.)

LENGTH is an integer.

CHARACTER is a character.

FILLCHAR fills DESTINATION with LENGTH instances of the character CHARACTER.

This could also be done (arcanely) by:

```
A[0] := <character>;
MOVELEFT(A[0],A[1],LENGTH-1);
```

... but FILLCHAR is twice as fast, as no memory reference is needed for a source. It is also more readable!

If DESTINATION is subscripted, FILLCHAR begins filling from the subscripted element.   The same applies if DESTINATION is a record with a field specification.

FILLCHAR is a **dangerous** intrinsic, and does no checking.   Use it with caution.

GOTOXY

Users' Manual
UCSD Pascal

V1.2.11 PROCEDURE GOTOXY(XCOORD , YCOORD: INTEGER);

This procedure sends the CONSOLE:'s cursor to the coordinates specified by (XCOORD, YCOORD).   The upper left corner of the screen is assumed to be (0,0).

A version of GOTOXY meant to run on all terminals is shipped with the System, but it is very slow.   An effective GOTOXY is one tailored to a particular type of terminal.   The user may write a GOTOXY and bind it in to the System using the utility LIBRARY.   More information on this is given in the Installation Guide.

VI.2.12 PROCEDURE HALT;

This procedure generates a HALT, which causes a runtime error to occur. The effect is similar to hitting the <break> key while a program is running. The console displays an error message saying that the program has halted itself: 'Programmed HALT or user break'.

If the DEBUGGER utility (see Section X.6) is active while a program is running, a HALT transfers control to the DEBUGGER, and allows the user to inspect or modify the state of the program.

INSERT

VI.2.13 PROCEDURE INSERT (SOURCE , DESTINATION , INDEX)

This inserts the string SOURCE into the string DESTINATION at the INDEXth
position in DESTINATION.

Example:

```
ID  := 'INSERTIONS';
MORE  := ' DEMONSTRATE';
DELETE(MORE,LENGTH(MORE),1);
INSERT(MORE,ID,POS('IO',ID));
WRITELN(ID);
```

Will print:

    INSERT DEMONSTRATIONS

VI.2.14   FUNCTION IORESULT : INTEGER;

After any I/O operation, IORESULT returns an INTEGER value corresponding to the values shown below.

Usually, the Compiler generates test code after each I/O operation, to see if the operation has failed, and abort the program if it has.   I/O checking may be turned off -- see Section VI.3.   Rather than allowing the program to fail, the user may wish to turn off I/O checking, and use IOCHECK to see if an I/O operation has failed; if it has, the program itself may take corrective action (such as re-displaying a prompt, for example).

Since any I/O operation will affect IORESULT (unless checking is turned off), doing a WRITELN(IORESULT) is not an informative diagnostic.   The following code achieves the desired effect:

```
    CHECK_RESULT := IORESULT;
    WRITELN(CHECK_RESULT);
```

I/O checks are never generated for the procedures UNITREAD or UNITWRITE.

These are the possible values of IORESULT:

```
     1 = parity error (CRC)
     2 = illegal device #
     3 = illegal IO request
     4 = data-com timeout
     5 = vol went off-line
     6 = file lost in dir
     7 = bad file name
     8 = no room on vol
     9 = vol not found
    10 = file not found
    11 = dup dir entry
    12 = file already open
    13 = file not open
    14 = bad input information
    15 = ring buffer overflow
    16 = write protect
    17 = illegal block
    18 = illegal buffer
```

The IORESULT value is stored in a single System-wide variable.   This means that

the user should be careful if there are concurrent processes which use IORESULT: the I/O performed by one process could change the information expected by another. The user should be extremely cautious with processes that are synchronized by attached semaphores, since their behavior is hard to predict; in other multiprocess situations, switching occurs at explicit SIGNAL and WAIT points, and problems with IORESULT are easily avoided. I/O done by the System itself does not affect a user program's IORESULT.

The table of IORESULT values also appears in Appendix B.

VI.2.15 FUNCTION LENGTH (SOURCE: STRING) : INTEGER

Returns the integer value of the dynamic length of SOURCE.

Example:

```
GEESTRING := '1234567';
WRITELN(LENGTH(GEESTRING),'   ',LENGTH(''));
```

Prints:

    7   0

MARK

Users' Manual
UCSD Pascal

VI.2.16   PROCEDURE MARK (VAR HEAPPTR: ^INTEGER)

Allocates a Heap Mark Record (HMR) on top of the Heap.

HEAPPTR must be a pointer. It is conventional to make it a ^INTEGER. The HMR contains valuable System information, so HEAPPTR must not be used as a pointer to available data space. To allocate memory, use the standard Pascal procedure NEW, or the UCSD intrinsic VARNEW.

See the Internal Architecture Guide for more details.

VI.2.17   FUNCTION MEMAVAIL: INTEGER;

Returns the number of unallocated words in memory. This is:

    (the number of words between the Code Pool and the Stack) plus
    (the number of words available in the Heap).

Note that MEMAVAIL does not return the <u>available</u> memory space, since there may be segments in main memory that could be overwritten if necessary. The intrinsic VARAVAIL should be used to determine space availability.

See the <u>Internal Architecture Guide</u> for more details.

MEMLOCK

Users' Manual
UCSD Pascal

VI.2.18   PROCEDURE MEMLOCK (SEGLIST: STRING)

SEGLIST must contain a list of segment names, separated by commas.

Loads the designated segments and "locks" them into main memory.

See the <u>Internal Architecture Guide</u> for more details.

VI.2.19 PROCEDURE MEMSWAP (SEGLIST: STRING)

SEGLIST must contain a list of segment names, separated by commas.

Swaps the designated (locked) segments out to disk.

See the Internal Architecture Guide for more details.

MOVELEFT

VI.2.20   PROCEDURE MOVELEFT (SOURCE, DESTINATION, LENGTH);

SOURCE and DESTINATION are any sort of arrays.   Or (as with
BLOCKREAD/WRITE and FILLCHAR) they may be of any other type, as well.   If
either is an array, it may be subscripted, and if either is a record, it may have a
field specification.

LENGTH is an integer.

Moves LENGTH bytes from SOURCE into DESTINATION, starting at the left.

MOVELEFT and MOVERIGHT are implemented as fast intrinsics.   They are rapid,
but do no range checking.   **BE CAREFUL!**

The rationale for having both MOVELEFT and MOVERIGHT is to facilitate the
transfer of bytes within a single array.

See MOVERIGHT for an example.

Vl.2.21   PROCEDURE MOVERIGHT (SOURCE, DESTINATION, LENGTH);

SOURCE and DESTINATION are any sort of arrays.   As with MOVELEFT, they may be any other type, as well.   Either may have a subscript (or if declared as a record, a field specification).

LENGTH is an integer.

Moves LENGTH bytes from SOURCE into the DESTINATION, starting at the right (the last byte, then the next-to-last, etc.).

This procedure is the counterpart to MOVELEFT.   It does no range checking, so **use caution.**

Example of both MOVELEFT and MOVERIGHT:

        VAR ARAY: PACKED ARRAY [1..30] OF CHAR;

                (*123456789a123456789b123456789c*)
            ARAY: THIS IS THE TEXT IN THIS ARRAY

MOVERIGHT(ARAY[10],ARAY[1],10);
            ARAY: NE TEXT INE TEXT IN THIS ARRAY

MOVELEFT(ARAY[1],ARAY[3],10)
            ARAY: NENENENENENETEXT IN THIS ARRAY

MOVELEFT(ARAY[23],ARAY[2],8);
            ARAY: NIS ARRAYENETEXT IN THIS ARRAY

POS

VI.2.22   FUNCTION POS (STRING , SOURCE) : INTEGER

STRING and SOURCE are string variables or constants.

POS attempts to match STRING to a substring of SOURCE. If STRING is matched, POS returns the location of the first character of the matched pattern. If STRING is not matched, POS returns zero.

Example:

```
STUFF  := 'TAKE THE BOTTLE WITH A METAL CAP';
PATTERN := 'TAL';
WRITELN(POS(PATTERN,STUFF));
```

... prints:

    26


```
PATTERN := 'CZECHOSLOVAKIA';
WRITELN(POS(PATTERN,STUFF));
```

... prints:

    0

VI.2.23 FUNCTION PWROFTEN (EXPONENT: INTEGER) : REAL;

This function returns the value of 10 to the EXPONENT power.

The legal range of EXPONENT varies from implementation to implementation, depending on the representation of floating point numbers. See the Installation Guide or the supplement for your hardware to find the limits.

REDIRECT

VI.2.24   FUNCTION REDIRECT (EXEC_OPTIONS: STRING): BOOLEAN

EXEC_OPTIONS is an execution option string as defined in Section 11.3.   It should contain only option specifications, and not the name of a file to execute (to execute a program from another program, see the CHAIN intrinsic).

REDIRECT causes redirection by performing all the options specified in EXEC_OPTIONS.   If all goes well, it returns TRUE.   If an error occurs, it returns FALSE.

If an error occurs during a call to REDIRECT, the state of redirection is indeterminate; this is a dangerous condition.   If REDIRECT returns FALSE, the user's program should follow it with a call to EXCEPTION, in order to turn off all redirection.   If the user does not do this, the results are unpredictable.   See the intrinsic EXCEPTION.

REDIRECT is a procedure in the Operating System's COMMANDIO unit; to use it, a program or unit must contain the declaration 'USES COMMANDIO'.

More information about redirection may be found in Section 11.3.

VI.2.25 PROCEDURE RELEASE (VAR HEAPPTR: ^INTEGER);

Cuts back the Heap from the current Heap Mark Record (HMR) to the HMR designated by HEAPPTR (HEAPPTR <u>must</u> have been initialized by a call to MARK).

MARKs and RELEASEs <u>must</u> be matched properly. See the warning in Section VI.1.2.3.1.

See the <u>Internal Architecture Guide</u> for more details.

SCAN

Users' Manual
UCSD Pascal

VI.2.26 FUNCTION SCAN (LENGTH, PARTIAL EXPRESSION, ARRAY): INTEGER;

LENGTH is an integer, ARRAY is usually a PACKED ARRAY OF CHAR, and PARTIAL EXPRESSION is a '<>' or a '=' followed by a single character in quotes.

SCAN scans ARRAY for LENGTH characters, <u>or</u> until it finds a character that satisfies the PARTIAL EXPRESSION (whichever comes first). It returns the offset from the position in ARRAY to the point at which it stopped.

In other words, if the position in ARRAY at which SCAN starts satisfies the PARTIAL EXPRESSION, SCAN returns zero. If the PARTIAL EXPRESSION is <u>not</u> satisfied, SCAN returns LENGTH. And if the PARTIAL EXPRESSION is satisf<u>ied</u> at some intermediate location, SCAN returns the offset from the starting position to that location.

LENGTH may be negative, in which case SCAN will scan from right to left, and return a negative value.

ARRAY may be subscripted, in which case SCAN will scan starting at that location. (ARRAY may in fact be of any type, but the user should exercise caution.)

Examples:

    DEM := '.....THE TERAK IS A MEMBER OF THE PTERODACTYL FAMILY.';

    SCAN(-26,=':',DEM[30]);

                            ... returns -26

    SCAN(100,<>'.',DEM);

                            ... returns 5

    SCAN(15,=' ',DEM[0]);

                            ... returns 8

VI.2.27 PROCEDURE SEEK (FILEID, INDEX);

FILEID is a file of any "structured" type. A file is considered structured if it is <u>not</u> a textfile (i.e., TEXT, INTERACTIVE, or FILE OF CHAR) or an untyped file.

INDEX is an integer.

SEEK moves the file window F^ to the INDEXth record in FILEID (indexing from zero). EOF and EOLN are set to False.

A GET or PUT <u>must</u> be made between two SEEKs, otherwise the window contents will be unpredictable.

SEMINIT

VI.2.28   PROCEDURE SEMINIT (VAR SEM: SEMAPHORE; SEM_COUNT: INTEGER)

Initializes the semaphore SEM to the value SEM_COUNT and establishes an empty queue.   See Chapter IX.

Warning! Failure to initialize a semaphore before using it in a SIGNAL or WAIT will put the System in an indeterminate state.

VI.2.29   PROCEDURE SIGNAL (VAR SEM: SEMAPHORE)

SIGNALs the semaphore SEM.

If no processes are waiting for SEM, the count associated with SEM is simply incremented. If one or more processes are waiting for SEM, then SEM is <u>not</u> incremented, and the process at the head of SEM's queue (the process with <u>the</u> highest priority) is added to the ready queue, where it competes with other ready processes for processor time. See Chapter IX.

SIZEOF

V1.2.30   FUNCTION SIZEOF (VARIABLE OR TYPE IDENTIFIER): INTEGER;

Returns the number of bytes that the variable or type has been allocated.

This is particularly useful for use as a parameter to FILLCHAR, MOVELEFT, or MOVERIGHT.

VI.2.31   PROCEDURE START (PROCESS CALL
[, PROCESSID
[, STACKSIZE
[, PRIORITY]]])

Starts a process.

PROCESS CALL is a call to a process -- it looks the same as a call to a procedure.   PROCESSID is a variable of type PROCESSID, STACKSIZE is an integer, and PRIORITY is in the range [0..255].   These latter three parameters are optional.

If no STACKSIZE parameter appears, the default STACKSIZE is 200.   If no PRIORITY parameter appears, the process's priority is the same as the priority of the parent process (i.e., the process that calls START).

Examples:

```
start(PLOP);
start( RED(I,J), PID);
start(FRAMUS(4), ID, 500);
start( RED(6,21), PID, SSIZE, 10);
```

Process declarations are similar to procedure declarations.   See Chapter IX.

Every process invocation (i.e., every call to START) is assigned a processid. This parameter, if present, is set to the processid value.   Processid's are intended for the System's use.

STACKSIZE, if present, allocates stack space to the process.   STACKSIZE defaults to 200 words.   A process needs 4 words + the amount occupied by local variables + room for activation records of procedures started by the process + (on most machines) space for the evaluation stack.   If a process is allocated less memory than it needs, the program will die with a stack overflow.

PRIORITY, if present, specifies the priority of the process.   Priorities determine (1) the ordering of a queue waiting for a semaphore, and (2) the ordering of the "ready queue" (i.e., the queue of all processes that are ready to run).   Priority defaults to the priority of the START'ing process.

STR

V1.2.32   PROCEDURE STR (LONG, DESTINATION)

LONG is either an integer or a long integer.   DESTINATION is a string.

STR converts LONG into a string, and places it in DESTINATION.   This intrinsic is chiefly used to format long integers for output.

See Section V1.1.2.4.5 for more about Long Integers.

Example:

```
INTLONG := 102039503;
STR(INTLONG,INTSTRING);
INSERT('.',INTSTRING,PRED(LENGTH(INTSTRING)));
WRITELN('$',INTSTRING);
```

... prints:

$1020395.03

VI.2.33   PROCEDURE TIME (VAR HIWORD, LOWORD: INTEGER);

Returns the value of the system's clock in 60ths of a second, stored in HIWORD_LOWORD as one 32-bit unsigned integer.   (Beware of LOWORD being treated as a negative two's-complement integer!)

TIME is dependent on there being a hardware system clock, and 16-bit integers, therefore it may not be available on all systems.

No conventions exist that would allow the user to treat the value returned by TIME as the time of day.   TIME is usually used for incremental time measurements, such as calculating benchmarks for a program.

UNITBUSY

V1.2.34   FUNCTION UNITBUSY (UNITNUMBER) : BOOLEAN;

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section 111.3).

UNITBUSY returns a Boolean value indicating whether the device UNITNUMBER is waiting for an I/O transfer to complete.

Example:

```
UNITREAD(2{non-echoing keyboard},CH[0],
          1{for one character},{no block no.},1{asynchronous});
WHILE UNITBUSY(2){While the READ has not been completed} DO
   WRITELN(OUTPUT,'1 am waiting for you to type something');
WRITELN(OUTPUT,'Thank you for typing a ',CH[0]);
```

... this program fragment will continuously type out the line '1 am waiting for you to type something' until a character is struck on the keyboard. Suppose a '!' were typed. The message 'Thank you for typing a !' will then appear, and program execution will proceed normally.

Currently implemented only on PDP-11 and LSI-11 computers. On all other systems, it always returns FALSE.

VI.2.35   PROCEDURE UNITCLEAR (UNITNUMBER);

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section lll.3).

UNITCLEAR cancels all l/Os to the specified unit and resets the hardware to its power-up state.

The function lORESULT can be used to determine if an error occured (this must be explicitly checked by the program, it will not be generated automatically). lORESULTs are listed in this section (see lORESULT) and Appendix B.

UNITREAD

Users' Manual
UCSD Pascal

VI.2.36   PROCEDURE UNITREAD
                  (UNITNUMBER, ARRAY, LENGTH,
                  [BLOCKNUMBER], [INTEGER]);

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section lll.3).

UNITREAD reads LENGTH bytes from the device UNITNUMBER into ARRAY (ARRAY may be of any type, but is usually a PACKED ARRAY OF CHAR).

UNITREAD is a low-level intrinsic, and should be used with extreme caution. It performs no I/O checking of any sort, and receives all characters sent by the device, including protocols, blank-compressions, and the like.

ARRAY may be subscripted, in which case it will be filled starting from that element.   CAUTION: some machines require that the starting position lie on a word boundary, so a program may not work on all machines if it reads bytes on an odd boundary.

BLOCKNUMBER is only meaningful if UNITNUMBER is a block-structured device. If it is present, it is the number of the block (zero-based) from which the read will start.   BLOCKNUMBER defaults to zero.

If INTEGER is present and equal to one, the transfer is done asynchronously (provided the hardware can support this).   INTEGER defaults to zero, which indicates a synchronous transfer.   If you wish to include the INTEGER parameter without the BLOCKNUMBER parameter, simply separate it from LENGTH by two commas, rather than one.

Example:

    UNITREAD(6,FILLME,80,,1)

... reads 80 characters asynchronously from REMIN: into FILLME; FILLME must be e'; least 80 characters long, or other data will be destroyed.

Because it references a device directly, input from UNITREAD cannot be redirected.

226

VI.2.37   PROCEDURE UNITSTATUS (UNITNUMBER, STATUS_REC, CONTROL)

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section 111.3).

STATUS_REC may be of any type; it should be an area of 30 words.

CONTROL is an integer. It should be equal to either 0 or 1.

UNITSTATUS returns information in STATUS_REC. If CONTROL=0, the information refers to output. If CONTROL=1, the information refers to input.

If the device in question is a character-oriented device (such as PRINTER:, REMOUT:, etc.), UNITSTATUS changes only the first word of STATUS_REC, and sets it equal to the number of characters waiting to be read or written. If there are no characters waiting, or UNITSTATUS cannot determine the device's state, it returns 0.

If the device is a block-structured device (such as a floppy disk), UNITSTATUS changes the first four words of STATUS_REC:

> word 1: The number of characters waiting (as with a serial device);

> word 2: The number of bytes per sector on the device;

> word 3: The number of sectors per track;

> word 4: The number of tracks.

While the remainder of STATUS_REC is not affected, these locations are reserved for possible future use.

UNITSTATUS is available on Adaptable Systems, but not a: other implementa   s; see the <u>Installation Guide</u> or the supplement for your hardw re for details.

UNITWAIT

VI.2.38 PROCEDURE UNITWAIT (UNITNUMBER);

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section 111.3).

UNITWAIT waits for the specified device to complete the I/O in progress. It can be simulated by:

WHILE UNITBUSY(n) DO {waste a small amount of time};

Currently implemented only on PDP-11 and LSI-11 computers. On all other systems, it always returns FALSE.

VI.2.39   PROCEDURE UNITWRITE
(UNITNUMBER, ARRAY, LENGTH,
[BLOCKNUMBER], [INTEGER]);

UNITNUMBER is an integer that is the number of a device (see Section 1.2.3 or Section 111.3).

UNITWRITE writes LENGTH characters from ARRAY to the device UNITNUMBER.

As with UNITREAD, no I/O checking is done, nor are any of the transmission's characters added or modified.   UNITWRITE is a low-level intrinsic, therefore it is fast but <u>dangerous.</u>

ARRAY may have a subscript, in which case the transfer will begin with that element.   As with UNITREAD, word boundaries must be enforced on some processors.

BLOCKNUMBER applies only to block-structured devices, and if present, indicates the number of the block (zero-based) where the write will start. BLOCKNUMBER defaults to zero.

INTEGER, if present and equal to one, indicates an asynchronous transfer. INTEGER defaults to zero.   If you wish to include INTEGER but not BLOCKNUMBER, separate it from LENGTH by one rather than two commas (as with UNITREAD).

Because it references a device directly, output from UNITWRITE cannot be redirected.

VARAVAIL

Users' Manual
UCSD Pascal

VI.2.40 FUNCTION VARAVAIL (SEGLIST) : INTEGER

SEGLIST is a string containing a list of segment names separated by commas. If a segment name is not recognized by the Operating System, it is ignored.

VARAVAIL returns the number of words in main memory available for allocation, assuming the segments listed are in memory, and all memory-locked segments are in memory.

See the Internal Architecture Guide for more details.

230

VI.2.41   PROCEDURE VARDISPOSE (POINTER, COUNT)

Does a DISPOSE on COUNT words. If COUNT is incorrect, VARDISPOSE will destroy the Heap's integrity, so use <u>extreme</u> caution.

POINTER is to an arbitrary pointer type.

See the <u>Internal Architecture Guide</u> for more details.

VARNEW

V1.2.42 FUNCTION VARNEW (POINTER, COUNT): INTEGER

Does a NEW on COUNT words.

POINTER is to an arbitrary type.

VARNEW returns the number of words actually allocated: if COUNT words are available, VARNEW should return COUNT, but if COUNT words are not available, VARNEW returns 0, and no words are allocated.

See the Internal Architecture Guide for more details.

VI.2.43   PROCEDURE WAIT (VAR SEM: SEMAPHORE)

If the count of SEM is greater than zero, it is decremented.   The process that called WAIT continues.

If the count of SEM is zero, the process using WAIT waits until SEM is again available.

See Chapter IX for examples.

V1.3   Using the Compiler

The UCSD Pascal Compiler is a recursive descent based on the P2 portable compiler from the Eidgenossische Technische Hochschuele, Zurich. It is invoked by using the C(ompile or R(un command of the outermost level of the System. If a workfile exists, it is compiled.   Otherwise, the user is prompted for a source file name.   The Compiler generates P-code, which is executed by some P-machine emulator (whether hardware or software).

While the Compiler is running, it displays a report of its progress on the CONSOLE:.

Example:

```
Pascal compiler - release level 1V.0
<    0>...................
INITIALIZE
<  19>........................................
GOOFOFF
<  61>...................................................
< 111>........
TEST
< 119>........................................

   237 lines compiled

INITIAL1 ....
TEST      .
```

In the first pass, the Compiler displays the name of each routine: the numbers enclosed within < > are the current line numbers, and each dot on the screen represents 1 source line compiled.   In the second pass, each name is the name of a segment, and each dot represents one routine.

This output can be suppressed in two ways -- for a given compilation, by using the Q+ Compiler option (see Section V1.3.1 below), or by setting HAS SLOW TERMINAL to TRUE in SYSTEM.MISCINFO (see the section on SETUP in the Installation Guide).

If the compilation is successful, that is, if no compilation errors were detected, the Compiler writes a codefile called *SYSTEM.WRK.CODE.   This is the codefile which may be executed by the R(un command.   See Chapter 11 for a description of System commands.

When the Compiler detects a syntax error, the text surrounding the error is displayed, along with an error number and '<<<< ' pointing to the offending symbol.

If both the Q and L options are set, the compilation continues, the syntax error is reported in the listing file, and CONSOLE: remains undisturbed.

If Q and L are not both on (this is the default) the Compiler gives the user the option of typing a space, an <esc> or 'E'. Typing a space continues the compilation, <esc> terminates the compilation, and 'E' calls the Editor, which places the cursor at the symbol where the error was detected.

The syntax errors detected by the UCSD Pascal Compiler are listed in Appendix 1. All error numbers are accompanied by a textual message after entering the Editor, provided *SYSTEM.SYNTAX is on disk. This error message also appears in the listing.

VI.3.1   Compile-Time Options

The user may direct some of the Compiler's actions by the use of compile-time options embedded in the source code.   Compile-time options are a set of commands that may appear within "pseudo-comments."   A pseudo-comment is a comment with a dollar sign immediately following the left-hand delimiter, for example:

```
{$I+}
(*$U MOLD.CODE*)
{$I+,S-,L+}
(*$R^*)
```

... are all correct pseudo-comments.   There are two kinds of options: "switch" options, and "string" options.   A switch option is a letter followed by a '+', '-', or '^'; a string option is a letter followed by a string.   A pseudo-comment may contain any number of switch options (separated by commas), and zero or one string options.   If a string option is present in a pseudo-comment, it must be the <u>last</u> option: the string is delimited by the option letter and the end of the comment.

If the pseudo-comment uses '(* *)' brackets, the string in a string option may <u>not</u> contain a '*'.

Some options may appear anywhere within a source file; others must appear at the front of the file (before the reserved word PROGRAM or UNIT).

String options use the string given them.   Switch options are either toggles or stack options.   If a switch option is a toggle, a '+' turns it on, and a '-' turns it off.   The options I and R are "stack options", as are the conditional compilation flags (see below).

With each stack option, the current state (<u>either</u> '+' or '-') is saved on the top of a stack (up to 15 states deep).   The stack may be "popped" by a '^' (thus re-enabling the previous state of that option).   If the stack is "pushed" deeper than 15 states, the <u>bottom</u> state of the stack is lost; if the stack is popped when it is empty, the value is always '-'.

Example:

>   {$I-} ... current value is '-' -- no I/O checking
>   ...
>   {$I+} ... current value is '+'
>   ...
>   {$I^} ... current value is '-' again
>   ...
>   {$I^} ... current value is '+', because this was the default
>   {$I^} ... current value is '-', because stack is now empty

Individual options are listed below in alphabetical order. The default options for a compilation are:

>   {$R+,I+,L-,U+,P+}

... these remain in effect unless the user overrides them. The Q option defaults to the opposite of the HAS SLOW TERMINAL data item in SYSTEM.MISCINFO (see the Installation Guide).

Conditional compilation is also controlled by compile-time options: see Section VI.3.2 below.

B:

B is a string option. Begins a section of conditionally compiled source code. See Section VI.3.2.

C:

C is a string option. Places all of the string directly into the copyright field of the codefile's segment dictionary. The purpose of this is to have a copyright notice embedded in the codefile.

D:

D is a string option. Used to declare or alter the value of a conditional compilation flag. See Section VI.3.2.

E:

E is a string option. Ends a section of conditionally compiled source code. See Section V1.3.2.

1:

There are two options named by '1': one is a stack switch option (IOCHECK), and one a string option (INCLUDE).

IOCHECK OPTION

Default value: 1+

1+: instructs the Compiler to generate code after each I/O
   statement, in order to check that the I/O operation
   was successful. If not, the program terminates
   with a runtime error.

1-: instructs the Compiler not to generate any I/O checking
   code. In the case of an unsuccessful I/O operation, the
   program merely continues.

The 1- option is useful when the user wishes to test IORESULT (see V1.2) when there is the chance of an I/O failure but the program should not be aborted. If 1- is used and the programmer does <u>not</u> test IORESULT, the effects are unpredictable.

IORESULTS are listed in Section V1.2 and Appendix B.

INCLUDE FILE MECHANISM

The string (delimited by the letter '1' and the end of the comment) is interpreted as the name of a file. If that file can be found, it is included in the source file and compiled.

Example:

       {$1 FOON}

... includes the file FOON.TEXT in the program's source.

If the initial attempt to open the include file fails, the Compiler concatenates a '.TEXT' to the filename and tries again. If this second attempt fails, or an I/O error occurs while reading the include file, the Compiler responds with a fatal syntax error.

In order that included source may carry its own declarations, an include file may contain CONST, TYPE, and VAR declarations, optionally followed by routine declarations. If this is the case, then the {$I...} comment must precede any routine declarations in the main program. Otherwise, the include file must follow normal Pascal ordering.

Include files may be nested up to three files deep (but no deeper).

Note that if a filename begins with a '+' or '-', a blank must be inserted between the letter I and the string.

Example:

        (*$I +FARKLE.STUFF*)

L:

L may be used either as a toggle switch option or a string option.

The default is L-. An L+ enables listing. If no listing file is named, the Compiler writes to *SYSTEM.LST.TEXT. The user may specify a different name for a listing file by using L as a string option, for example:

        (*$L DEMO1.TEXT*)

... this would write to DEMO1.TEXT on the default disk.

Note that listing files which are sent to the disk may be edited as any other textfile, provided they are created with a .TEXT suffix. Without the .TEXT suffix, the System will treat the listing as a datafile.

Here is a sample portion of a listing:

```
393 10    10:D      1 Procedure iocheck;
{ commented out ';' }{;
{ commented out ';' } This procedure will check the i/o operations of the
{ commented out ';' } index as it is in the process of rebuilding
397 10    10:D      1 }
398 10    10:0      0 Begin
399 10    10:1      0    If ioresult <> 0 Then
400 10    10:2      6      Begin
401 10    10:3      6        pl^ := 'index l/O failure.';
402 10    10:3     32        prompt(errorline);
403 10    10:2     38      End; { if ioresult <> 0 then }
404 10    10:0     38 End; { iocheck }
405 10    10:0     50
406 10    11:D      1 Procedure dropindex(position: isamcoverage);
```

On those lines that are not {commented out ';'} (which is intended to warn you that a comment may have accidentally swallowed some Pascal code), the numbers that precede a source line are:

the line number,

the segment number,

the routine number:lex level, and

the number of words of data or code storage
which the routine requires at that point.

Rather than a lexical level, lines of declarations show a 'D' following the procedure number.

Here is a portion of a listing with errors:

```
  596 10    1:5    228        lastpageitem := min(lastentry,lastentry);
---> Error #104 <---
  597 10    1:5    239
  598 10    1:5    239        { loop through the page }
  599 10    1:5    239        PageInx := 0;
  600 10    1:5    242        { function returns next greater }
  601 10    1:5    242        Repeat {until found or (PageInx > lastentry)}
  602 10    1:6    242          Assert(PageInx < lastpageitem,'bad PageInx');
---> Error #104 <---
    The previous error is on line 596
  607 10    1:6    271          found := (data[PageInx].key > key);
```

Error messages indicate the position of the previous error. The Compiler also lists readable error messages from *SYSTEM.SYNTAX, provided it is on line.

Whether or not the compilation is completed, the listing is saved.

P:

P is a switch option. P- turns off pagination in the listing, P+ turns it on again, and a P by itself starts a new page in the listing.

Q:

The 'quiet compile' option. This is used to suppress the Compiler's output to CONSOLE:.

Default value:
    the value of SLOWTERM in *SYSTEM.MISCINFO
    (see SETUP in the Installation Guide).

Q+: causes the Compiler to suppress output to CONSOLE:.

Q-: causes the Compiler to send progress information
    to CONSOLE:.

R:

R is a stack switch option.

Default value: R+

R+: turns range checking on.

R-: turns range checking off.

Programs compiled with the R-option set will run slightly faster; however, if an invalid index occurs or a invalid assignment is made, the program will not be terminated with a runtime error, and the results are exceedingly hard to debug. Until a program has been completely tested, it is strongly advised to compile with the R+ option left on.

T:

T is a string option. The string becomes the new title of pages in the listing file.

U:

There are two options indicated by U: one is a toggle switch option (USER PROGRAM), and one is a string option (USE LIBRARY).

USER PROGRAM OPTION

This option determines whether this compilation is a user program compilation, or a compilation of a System program. If present, it must appear before the heading (i.e., before the reserved word PROGRAM or UNIT).

Default value: U+

U+: specifies user source.

U-: allows compilation of units with names that are predeclared in the System. Also sets R- and I-.

The average user will never use this option, except when compiling GOTOXY (see the Installation Guide).

USE LIBRARY OPTION

This is a string option: the string is interpreted as a filename.

If the file named in the U option can be found, the Compiler will search it for the code of UNITs named in subsequent USES declarations. If a UNIT is not found there, the Compiler then searches *SYSTEM.LIBRARY.

If a program contains USES declarations but no U option, the Compiler will look for the USEd UNITs first in the source file itself, and in *SYSTEM.LIBRARY.

Following is an example of a valid USES clause using the 'U' option:

```
USES UNIT1,UNIT2, { Found in *SYSTEM.LIBRARY }
   {$U A.CODE}
      UNIT3,
   {$U B.LIBRARY}
      UNIT4,UNIT5;
```

V1.3.2   Conditional Compilation

Portions of source code may be conditionally compiled. Whether they are compiled or not depends on the value of a flag that is declared by a compile-time option at the beginning of the source file.

A section of source code to be conditionally compiled must be delimited by the options B and E. Both these options must name the flag which determines whether the code is compiled. The flag itself is declared by a D option at the beginning of the source; D options may be used at other locations in the source to change the value of an existing flag.

Example:

```
{$D DEBUG}   {declares DEBUG and sets it TRUE}
PROGRAM SIMPLE;
...
BEGIN
    ...
    {$B DEBUG}   {if DEBUG is TRUE, this section is compiled}
    WRITELN('There is a bug.');
    {$E DEBUG}   {this ends the section}
    ...

    ...
    {$B DEBUG-}   {if DEBUG is FALSE, this section is compiled}
    WRITELN('Nothing has failed.');
    {$E DEBUG}
    ...
END {SIMPLE}.
```

Each flag in a program <u>must</u> appear in a D option before the source heading. The name of a flag follows the rules for Pascal identifiers. If the flag's name is followed by a '-', that flag is set FALSE. The flag may be followed by a '+', which sets it TRUE. If no sign is present, a flag is TRUE. The flag's name may also be followed by a '^': see below.

The state of a flag may be changed by a D option which appears <u>after</u> the source heading, but if the flag has not been declared, an error will result.

The B and E options delimit a section of code to be conditionally compiled: when the Compiler encounters a B option, it scans for an E option which names the same flag, and resumes compilation from that point. The B option may follow the flag's name with a '-', which causes the delimited code to be compiled if the flag is FALSE. In the absence of a '-', the code is compiled if the flag is TRUE.

The flag's name may also be followed by a '+' or '^': these are ignored. In an E option, the flag's name may be followed by a '+', '-', or '^': these symbols are ignored.

The state of each flag is saved in a stack, just as the state of a stack switch option is saved. Thus, using a D option with '^' yields the previous value of the flag. Each flag's stack is 15 values deep. If a 16th value is pushed, the <u>bottom</u> of the stack is lost; if an empty stack is popped with '^', the value returned is always FALSE.

If a section of code is <u>not</u> compiled, then any pseudo-comments it may contain are ignored as well.

Example:

```
{$D DEBUG-}  {declares DEBUG and sets it FALSE}
PROGRAM SIMPLE;
...
BEGIN
    {$D DEBUG+}  {changes DEBUG to TRUE}

    {$B DEBUG}  {if DEBUG is TRUE, this section is compiled}
    WRITELN('There is a bug.');
    {$E DEBUG}  {this ends the section}
    ...

    {$D DEBUG^}  {restores previous value of DEBUG}
                 {... in this case, FALSE}
    {$B DEBUG-}  {if DEBUG is FALSE, this section is compiled}
    WRITELN('Nothing has failed.');
    {$E DEBUG}
    ...
END {SIMPLE}.
```

VII.   THE ADAPTABLE ASSEMBLERS

VII.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.

VII.0.2   Assembly Language Applications

Users of the UCSD Pascal 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 another
   high level language (such as FORTRAN or BASIC).

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 Pascal 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 document is intended to be used in conjunction with the processor software manual of the user's machine. For information concerning differences from the processor's standard software syntax, see Section VII.8 (all section references in this chapter refer to this chapter alone).

VII.1 General Programming Information

VII.1.1 Object Code Format

VII.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.

VII.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.

VII.1.1.3 Memory Organization

VII.1.1.3.1 Byte Versus Word Addressing

The Adaptable Assembler kernel is designed to accommodate byte addressed, word addressed, and byte addressed/word oriented processors - the instructions and data words of the latter two processor types are constrained to word boundaries. A word boundary on a word oriented processor is defined as an even byte address.

Word alignment is enforced throughout word addressed versions of the assembler by defining all data directives to emit integral numbers of words. With word oriented assemblers, the programmer is responsible for maintaining word alignment of instructions and data words; failure to do so will be flagged with an error message. Nonalignment occurs when a directive creates an odd number of data bytes.

VII.1.1.3.2 Byte Sex

The two bytes that make up a 16-bit word are termed the least-significant and most-significant byte, or LSB and MSB respectively. Unfortunately, the various

manufacturers of processors have different conventions for whether the first (or lower addressed) byte of a word is used as the LSB or the MSB; hence, the 'byte sex' question has arisen. See Section Vll.8 to determine the byte sex of processors supported with Adaptable Assembler versions.

The assemblers generate code of the proper byte sex, regardless of the sex of the machine on which they are run.

VII.1.2   Source Code Format

VII.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

VII.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.

VII.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. Each assembler also has a predefined location counter symbol. This is a character which, when used in an expression, represents the current value of the location counter in the program. Section VII.8 lists the location counter character for each assembler version.

VII.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.

VII.1.2.4 Constants

VII.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

VII.1.2.4.1 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.

VII.1.2.4.2   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.

VII.1.2.4.3   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

VII.1.2.4.4   Default Radix Integer Constants

The radix of an integer constant lacking a trailing radix character is set to the assembler's current default radix.   Initial default radices for all assemblers are listed in Section VII.8.

VII.1.2.4.5   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"

VII.1.2.4.6   Assembly time Constants

An assembly time constant is written as an identifier that has been assigned a constant value by the .EQU directive (Section VII.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.

VII.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.

VII.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.

VII.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.

VII.1.2.5.3   Arithmetic & 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.

Vll.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.

VII.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)

V11.1.3   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.

VII.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.

VII.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 VII.2.2) are assigned the value of the expression in the operand field.

VII.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
              LDA         3
$3            STA         4
              JP          NZ,$3  ; legal use of local label
              NOP
              JP          $4     ; illegal use
LABEL2
              LDA         5
$4            STA         6
```

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 V11.2.2).

VII.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

VII.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.

VII.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.

VII.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.

VII.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.
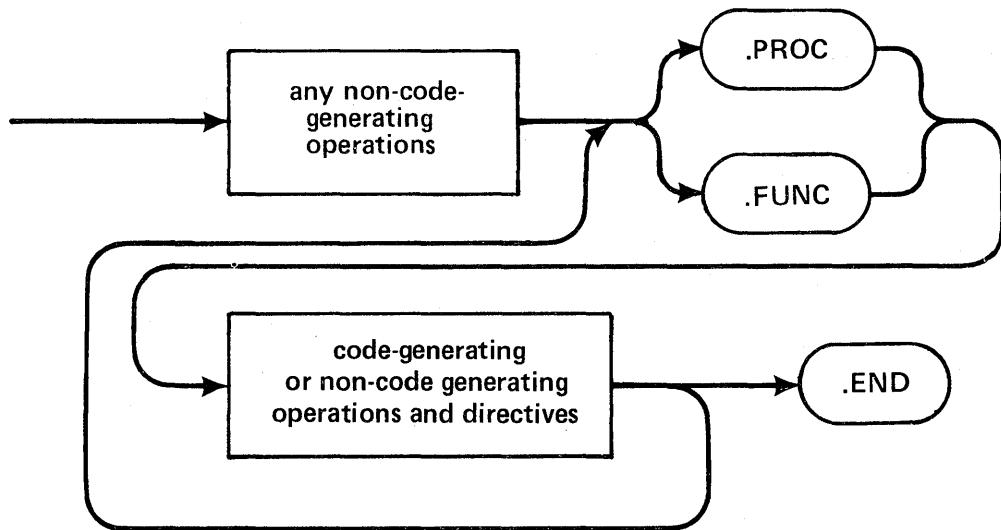
Assembly routines must begin with a .PROC, .FUNC, .RELPROC, or .RELFUNC directive.   The last routine in the source file must be terminated by the .END directive.   Section VII.5 gives a detailed description of these directives.

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.

VII.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.).

VII.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 locatio

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
```

VII.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 VII.1.2.4.

label =
>   any legal label as defined in Section VII.1.3.1.

expression =
>   any legal expression as defined in Section VII.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 Vll.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 Vll.1.3.4.

character string =
>   any legal character string as defined in
>   Section Vll.1.2.3.

file identifier =
>   any legal name for a Pascal text file.

Example:

>   [<label>] [b] .ASCll 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.

VII.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 VII.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

267

.RELPROC   Identifies the beginning of a dynamically
               relocatable assembly language procedure. Such
               assembly procedures must be position-independent
               (see Section VII.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

VII.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 hex

      TEMP1 .BYTE   ; code would be: 00 hex

.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

VlI.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

On a byte addressed processor, this would align the LC
on a word boundary.

VII.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
VII.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

274

.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 ASCll 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 V11.4
             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
             VII.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

VlI.2.5 Program Linkage Directives

Linking directives enable communication between separately assembled and/or
compiled programs.   Section VlI.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   1,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

VII.2.6   Conditional Assembly Directives

Section VII.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

Vll.2.7 Macro definition directives

Section Vll.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

VII.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 VII.5 has a
detailed description of absolute code files.

FORM: [b] .ABSOLUTE [<comment>]

EXAMPLE: .ABSOLUTE

.ASECT   Specifies the start of an absolute section. Section VII.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 VII.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 Vll.1.2.2.4 has a detailed description of
                radices.   Initial defaults for each assembler version
                are listed in Section Vll.8.

                FORM: [b] .RADIX <integer> [<comment>]

                EXAMPLE: .RADIX   10   ; decimal default radix

VII.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 VII.2) is as follows:

```
.IF <conditional expression>
<source statements>
[.ELSE
<source statements>]
.ENDC
```

VII.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).

Vll.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
```

VII.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"
```

VII.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.

VII.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.

VII.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 <u>not</u> 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
SRL     %2
.ENDM

.MACRO  UNO
MOV     %1 ,%2
SLA     %3
.ENDM
```

In a program, the macro call...

```
DOS     TROIS,DEUX
```

assembles as...

```
MOV     DEUX,UN     ; UNO got UN directly, but had to
                    ; use DOS's 2nd param
SLA                 ; 3rd param doesn't exist
SRL     DEUX        ; DOS used its own 2nd param
```

VII.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.

VII.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
      BEQ      $1
      BNE      %1
$1
      .ENDM
```

In a program, the code...

```
TWIE
      MOV      ICH1,N1
      EIN      $1
      RTS
$1
      JSR      SAN
```

assembles as...

```
TWIE
      MOV      ICH1,N1         ; looks confusing, but if listing
                               ; was off, result is what programmer
                               ; meant to occur
      BEQ      $1              ; this references macro local label
      BNE      $1              ; this references outside $1
$1                             ; macro local label

      RTS
$1                             ; outside $1
      JSR      SAN
```

VII.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.

VII.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.

VII.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 VII.6.2.1 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.

Vll.5.1.2 External Reference Directives

The directives .REF and .DEF allow separately assembled modules to share data space and subroutines. See Vll.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.

VII.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 VII.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.

VII.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.

VII.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 VII.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. Section VII.8 has a detailed description of Pascal/assembly language protocols for all machines.

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.

VII.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. Machine dependent data representations are described in another section of the user manual.

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'.

VII.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.

VII.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 Section VII.5.2.1.1.2.1).

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.

VII.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 (an implementation-dependent value) or a pointer to a segment environment record.

(In a future release, the use of NIL will be replaced by use of the value zero, in order to eliminate this implementation dependence.)

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.

307

VII.5.2.2   Example Of Linking To Pascal Host

Note that in the following example the host program passes control to the beginning of an assembly procedure whether or not machine instructions are present there; therefore, all data sections allocated in the procedure must either occur after the end of the machine instructions or have a jump instruction branch around them.

```
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 }

   procedure do_nothing; external;
   function null_func(xxyxx,z: integer): integer; external;

begin
  k := 45;
  do_nothing;
  j := null_func(k,size);
end.
```

```
        .PROC       DONOTHING  ; underscores are not significant
                               ; in Pascal

        .CONST      SIZE       ; can get at size constant in host...
        .PUBLIC     1,LST1     ; and also these two global vars
        .DEF        TEMP1      ; this allows NULLFUNC to get at temp1
                               ; code starts here...
        POP         RETADR     ; assume return addr pushed on stack

        ; does nothing

        PUSH        RETADR     ; set up stack for return
        RETURN
                               ; data area
        RETADR      .EQU  TEMP1
        TEMP1       .WORD
                               ; end of procedure DONOTHING


        .FUNC       NULLFUNC,2

        .PRIVATE    PRT,LST2:9 ; 10 words of private data
        .REF        TEMP1      ; references data temp in DONOTHING
                               ; code starts here

        POP         RETURN     ; save return address

        POP         PRT        ; get parameter 'z'
        POP         LST2+4     ; get parameter 'xxyxx'

        POP         TEMP1      ; toss 1 word of junk

        ;  performs null action

        PUSH        LST2+4     ; return xxyxx as result
        PUSH        RETURN     ; restore subr link
        RETURN                 ; return to calling program
                               ; data starts here...
RETURN  .WORD
                               ; end of assembly
        .END
```

VII.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 VII.5.2.3.2 provides some ideas on how to create a Pascal loader program.

The utility COMPRESSOR is a much easier and more versatile way of doing this task.   It allows for relocation and compaction of code.   Refer to Section X.1.

VII.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.

VII.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 possiblity 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.

VII.5.2.3.3 Byte Sex Considerations

All assembler versions and the System Linker are designed to produce assembly codefiles with the correct byte sex of the target processor, regardless of the byte sex of the machine underlying the Pascal system in use; thus, there is no need for 'code flipping' software.

VII.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'.   Assemblers that are not in use are usually saved with the file name 'ASM <processor #>.CODE' (e.g., 'ASM6809.CODE').

VII.6.1   Support Files

Each assembler version has two associated support file: 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 '<processor #>.OPCODES', where <processor #> matches the processor of the current system assembler.   The opcode file contains all predefined symbols (instruction 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.

Each assembler also has its own error file which contains a list of machine specific error messages.   The error file must have the name '<processor #>.ERRORS', where <processor #> matches the processor of the current system assembler. The presence of an error file is not necessary for running the assembler, but it can greatly aid the chore of squeezing   the syntax errors out of a freshly written program.

312

VII.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 VII.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.

VII.6.3 Responses To Listing Prompt

Before assembling begins, the following prompt will appear on the console:

xxxx   Assembler [yy]
Output file for assembled listing: (<CR> for none)

xxxx is the processor number and yy 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.

VII.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.

VII.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 VII.6.2) in order for the editor to position the
   cursor in the file that contains the error.

VII.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.

VII.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.

VII.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.

Vll.7.2   Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console (see Section Vll.6), except that the user prompt is not included.

VII.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 the default list radix of the assembler version being used   (either hex or octal - see Section VII.8). Separately emitted bytes and words of code on the same line are delimited   by spaces.

VII.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.

VII.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.

VII.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.

VII.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.

VII.7.5   Example

The following is a small example of an assembled listing:

```
0000|
0000|                            .ABSOLUTE
0000|
0000|                            .PROC     PRIMARYZ
0000|
0000|  0BFD        FLOPPY  .EQU  0BFDH        ;Rom-based floppy driver
0000|  9000        SECMEM  .EQU  9000H        ;First location in memory
0000|  9000        SECENT  .EQU  9000H        ;Entry point of bootstrap
0000|  1708        SECDSK  .EQU  08H + 1700H  ;Sector start of 2nd bootstrap
0000|  1710        B1DSK   .EQU  10H + 1700H  ;Sector start of BIOS part 1
0000|  1718        B2DSK   .EQU  18H + 1700H  ;Sector start of BIOS part 2
0000|
0000|                            .ORG  1000H       ;Primary boot for GRISWICH DOS
1000|
1000|  FD 21 ****  PRIMARY LD    1Y,SECREAD   ;Get block for second bootstrap
1004|  CD FD0B             CALL  FLOPPY
1007|  FD 21 ****          LD    1Y,B1READ    ;Get block for part 1 of BIOS
100B|  CD FD0B             CALL  FLOPPY
100E|  FD 21 ****          LD    1Y,B2READ    ;Get block for part 2 of BIOS
1012|  CD FD0B             CALL  FLOPPY
1015|  C3 0090             JP    SECENT       ;Jump into second bootstrap
1018|
1002*  1810
1018|              SECREAD
1018|  00                  .BYTE              ;Unused
1019|  0A                  .BYTE OAH          ;Read command
101A|  0090                .WORD SECMEN       ;Memory loc. for second boot
101C|  0002                .WORD 200H         ;Number of bytes in boot
101E|  0000                .WORD              ;Completion return address
1020|  0010                .WORD PRIMARY      ;Error in return address
1022|  00                  .BYTE              ;Completion result code
1023|  0817                .WORD SECDSK       ;Disk block of second boot
1025|
1009*  2510
1025|
1025|              B1READ
1025|  00                  .BYTE              ;Unused
1026|  0A                  .BYTE OAH          ;Read command
1027|  0093                .WORD SECMEN+300H  ;Memory location or BIOS part 1
1029|  0002                .WORD 200H         ;Number of bytes in BIOS part 1
```

```
102B| 0000                    .WORD              ;Completion return address
102D| 0010                    .WORD PRIMARY      ;Error return address
102F| 00                      .BYTE              ;Completion result code
1030| 1017                    .WORD B1DSK        ;Disk block of BIOS part 1
1032|
1010* 3210
1032|              B2READ
1032| 00                      .BYTE              ;Unused
1033| 0A                      .BYTE 0AH          ;Read command
1034| 0095                    .WORD SECMEN+500H  ;Memory location of BIOS part 2
1036| 0002                    .WORD 200H         ;Number of bytes in BIOS part 2
1038| 0000                    .WORD              ;Completion return address
103A| 0010                    .WORD PRIMARY      ;Error return address
103C| 00                      .BYTE              ;Completion result code
103D| 1817                    .WORD B2DSK        ;Disk block of BIOS part 2
103F|
103F|                         .END
```

PAGE- 2 PRIMARYZ FILE:#5:PRIMARY.Z SYMBOLTABLE DUMP

    AB - Absolute LB - Label   UD - Undefined MC - Macro
    RF - Ref   DF - Def   PR - Proc   FC - Func
    PB - Public   PV - Private CS - Constant


```
B1DSK     AB 1710|  B1READ  LB 1025|  B2DSK    AB 1718|  B2READ   LB 1032|
FLOPPY    AB 0BFD|  PRIMARY LB 1000|  PRIMARYZ PR --|   SECDSK    AB 1708|
SECENT    AB 9000|  SECMEM  AB 9000|  SECREAD  LB 1018|
```

VII.8   Machine-Specific Information

This section is intended to be used in conjunction with processor manuals distributed by the manufacturers of the various processors.   These manuals provide syntax conventions for the instruction sets and address modes used by the corresponding Adaptable Assembler versions.   The company chosen as a base for syntax conventions is listed for each version, along with a list of deviations from that company's syntax conventions.

VII.8.1   LSI-11/PDP-11 Assembler

VII.8.1.1   Syntax Conventions

The 11 assembler adheres to DEC standard syntax for opcode fields, register names, and address modes.   The location counter symbol is an asterisk '*'.

VII.8.1.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. Registers 0 and 1 are available to the assembly routine; other registers must be saved on entry and restored on exit.

VII.8.1.3   Memory Organization

The 11 processor is byte addressed and word oriented; machine instructions and data words must be aligned to start on an even byte boundary.   The byte sex is least-significant-byte-first.

VII.8.1.4   Default Constant and List Radices

The default constant radix and default list radix are octal.

VII.8.2   Z80 Assembler

VII.8.2.1   Syntax Conventions

The Z80 assembler adheres to Zilog standard syntax for opcode fields, register names, and address modes. The following conventions may deviate from this standard:

- the syntax for exchanging the register pair AF and
  the alternate register pair AF' is the following:

      EX   AF

The location counter symbol is a dollar sign '$'.

VII.8.2.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

VII.8.2.3   Memory Organization

The Z80 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

VII.8.2.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.3   6500 Assembler

VII.8.3.1   Syntax Conventions

The 6500 assembler adheres to Rockwell standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- immediate operands are specified by using a preceding pound sign '#' character:

```
LABEL    .EQU    5
  LDA    #LABEL          ; immediate
```

- zero-page addressing is achieved only by using absolute operands (i.e., assembly time constants) with values between 0 and 255:

```
LABEL    .EQU    5
  LDA    LABEL           ; zero-page
```

- indirect addressing has the following form:

```
  LDA    @LABEL,X     ; indexed-indirect (preindexing)
  LDA    @LABEL,Y     ; indirect-indexed (postindexing)
  JMP    @LABEL       ; indirect jump
```

The location counter symbol is an asterisk '*'.

VII.8.3.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

VII.8.3.3   Memory Organization

The 6502 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

VII.8.3.4   Default Constant and List Radices

The default constant radix and default list radix are hexadecimal.

VII.8.4   6800 Assembler

VII.8.4.1   Syntax Conventions

The 6800 assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- all instructions which can specify the A and B
  registers have the register name separated from
  the opcode field:

  ```
  LDA    A,LABEL
  LDA    A,0,X        (instead of LDA    A,X)
  LDX    0,X          (instead of LDA    X)
  STA    A,14,X
  PUL    A
  ASL    B
  ```

- immediate operands are specified by using a preceding
  pound sign '#' character:

  ```
  LABEL    .EQU    5
  LDA      A,#LABEL        ; immediate
  ```

- zero-page addressing is achieved only by using absolute
  operands (i.e., assembly time constants) with values
  between 0 and 255:

  ```
  LABEL    .EQU    5
  LDA      B,LABEL         ; zero-page
  ```

- numbers in hex must always contain four digits (yes, even
  for bytes):

  ```
  .BYTE    0002H,00A9H    specifies the quantity 02A9 base 16
  ```

The location counter symbol is an asterisk '*'.

VII.8.4.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

VII.8.4.3   Memory Organization

The 6800 processor is byte addressed and byte oriented. The byte sex is most-significant-byte-first.

VII.8.4.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.5   8080 Assembler

VII.8.5.1   Syntax Conventions

The 8080 assembler adheres to Intel standard syntax for opcode fields, register names, and address modes.   The location counter symbol is a dollar sign '$'.

VII.8.5.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

VII.8.5.3   Memory Organization

The 8080 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

VII.8.5.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.6   9900 Assembler

VII.8.6.1   Syntax Conventions

The 9900 assembler adheres to TI standard syntax for opcode fields, register names, and address modes.   The following conventions may deviate from this standard:

- in operand fields, the lack of an address mode
  character (i.e., a '@' or '*' preceding the operand)
  defaults to '@'.

The location counter symbol is a dollar sign '$'.

VII.8.6.2   Sharing of Machine Resources with Interpreter

The return address to the system is passed in register 11. Registers 0 thru 5 are available to the assembly routine; other registers must be saved on entry and restored on exit.

VII.8.6.3   Memory Organization

The 9900 processor is byte addressed and word oriented; machine instructions and data words must be aligned to start on an even byte boundary.   The byte sex is most-significant-byte-first.

VII.8.6.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.7   6809 Assembler

VII.8.7.1   Syntax Conventions

The 6809 Assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- immediate operands are specified by using a preceding '#':

    ```
    ANDCC          #01
    ```

- indirect addressing is specified by a single leading at-sign ('@') instead of square brackets ('[ ]'):

    ```
    LDX            @THERE,PCR
    ```

- zero-page addressing is achieved only by using operands that are absolute (i.e., not labels) and less than 256:

    ```
    ZEROPAGE   .EQU    15
        LDB      ZEROPAGE
    ```

VII.8.7.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.7.3   Memory Organization

The 6809 processor is byte-addressed and byte-oriented. The byte sex is most-significant-byte first.

VII.8.7.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.8   Z8 Adaptable Assembler

VII.8.8.1   Syntax Conventions

VII.8.8.1.1   Symbols

The Z8 Adaptable Assembler adheres to Zilog standard syntax (refer to the Z8 PLZ/ASM Assembly Language Programming Manual) for opcode fields, register names, and addressing modes.

VII.8.8.1.2   Numeric Constants

The Z8 Assembler follows the constant conventions of other Adaptable Assemblers, except that octal constants are indicated by a radix switch character of 'O' rather than 'Q', and binary constants are indicated by a radix switch character of 'B' rather than 'T'.

Example: 011101B   0B   1467O   11110O

VII.8.8.1.3   Predefined Constants

There are no predefined constants in the Z8 Assembler.   Specifically, the constants '%L', '%T', '%R', '%P', '%%', and '%Q' in Zilog syntax are NOT allowed.

VII.8.8.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.8.3   Memory Organization

The Z8 processor is byte-addressed and byte-oriented.   The byte sex is least-significant-byte-first.

VII.8.8.4   Default and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.7   6809 Assembler

VII.8.7.1   Syntax Conventions

The 6809 Assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- immediate operands are specified by using a preceding '#':

    ANDCC   #01

- indirect addressing is specified by a single leading at-sign ('@') instead of square brackets ('[ ]'):

    LDX   @THERE,PCR

- zero-page addressing is achieved only by using operands that are absolute (i.e., not labels) and less than 256:

    ZEROPAGE .EQU   15
       LDB   ZEROPAGE

VII.8.7.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.7.3   Memory Organization

The 6809 processor is byte-addressed and byte-oriented. The byte sex is most-significant-byte first.

VII.8.7.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.8   Z8 Adaptable Assembler

VII.8.8.1   Syntax Conventions

VII.8.8.1.1   Symbols

The Z8 Adaptable Assembler adheres to Zilog standard syntax (refer to the Z8 PLZ/ASM Assembly Language Programming Manual) for opcode fields, register names, and addressing modes.

VII.8.8.1.2   Numeric Constants

The Z8 Assembler follows the constant conventions of other Adaptable Assemblers, except that octal constants are indicated by a radix switch character of 'O' rather than 'Q', and binary constants are indicated by a radix switch character of 'B' rather than 'T'.

Example: 011101B   0B   1467O   111100

VII.8.8.1.3   Predefined Constants

There are no predefined constants in the Z8 Assembler.   Specifically, the constants '%L', '%T', '%R', '%P', '%%', and '%Q' in Zilog syntax are NOT allowed.

VII.8.8.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.8.3   Memory Organization

The Z8 processor is byte-addressed and byte-oriented.   The byte sex is least-significant-byte-first.

VII.8.8.4   Default and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.7   6809 Assembler

VII.8.7.1   Syntax Conventions

The 6809 Assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- immediate operands are specified by using a preceding '#':

    ANDCC   #01

- indirect addressing is specified by a single leading at-sign
  ('@') instead of square brackets ('[ ]'):

    LDX   @THERE,PCR

- zero-page addressing is achieved only by using operands that
  are absolute (i.e., not labels) and less than 256:

    ZEROPAGE   .EQU   15
       LDB   ZEROPAGE

VII.8.7.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.7.3   Memory Organization

The 6809 processor is byte-addressed and byte-oriented. The byte sex is most-significant-byte first.

VII.8.7.4   Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

VII.8.8   Z8 Adaptable Assembler

VII.8.8.1   Syntax Conventions

VII.8.8.1.1   Symbols

The Z8 Adaptable Assembler adheres to Zilog standard syntax (refer to the Z8 PLZ/ASM Assembly Language Programming Manual) for opcode fields, register names, and addressing modes.

VII.8.8.1.2   Numeric Constants

The Z8 Assembler follows the constant conventions of other Adaptable Assemblers, except that octal constants are indicated by a radix switch character of 'O' rather than 'Q', and binary constants are indicated by a radix switch character of 'B' rather than 'T'.

Example: 011101B   0B   1467O   11110O

VII.8.8.1.3   Predefined Constants

There are no predefined constants in the Z8 Assembler. Specifically, the constants '%L', '%T', '%R', '%P', '%%', and '%Q' in Zilog syntax are NOT allowed.

VII.8.8.2   Sharing of Machine Resources with Interpreter

No interpreter is currently available for the 6809.

VII.8.8.3   Memory Organization

The Z8 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

VII.8.8.4   Default and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

SAGE
COMPUTER TECHNOLOGY

Linker

p-System Supplement

VIII.   SEGMENTS, UNITS, and LINKING

VIII.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.

VIII.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 code 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.

The other high-level languages in the p-System use their own syntax for creating separate segments: refer to each particular language's manual for details.

VIII.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 VII 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 (see Section VI.3), and inform the Operating System by including the codefile's <u>name</u> in a "library text file." The default library text file is *USERLIB.TEXT, but can be changed by an execution option. See Sections VIII.3 and II.3.

The other high-level languages in the p-System use their own syntax for separate compilation: refer to each particular language's manual for details.

Vlll.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 VIII.2 and VIII.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.

Vlll.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}.
```

Vlll.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 (full syntax railroad diagrams may be found in Appendix H):

```
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.

342

An INTERFACE part is terminated by the UCSD reserved word IMPLEMENTATION.

An INTERFACE part may not contain $Include files (see Section VI.3).   An INTERFACE part may be contained <u>within</u> an $Include file, provided that <u>all</u> 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;                          UNIT GOLE2;
   INTERFACE                            {$1  INTER_PART}
      {$1  INTER_DECS}               IMPLEMENTATION
   IMPLEMENTATION                       ...
      ...                            END;
   END;
```

... are <u>not</u> legal forms of a UNIT, while the following outline <u>is</u>:

```
UNIT GOLE3;
{$1  WHOLE_UNIT}
;
```


The <initialization statements> and <termination statements> are optional sections of code. Initialization statements, if present, are executed <u>before</u> any of the code in a host that USES the UNIT is executed, and termination statements, if present, are executed <u>after</u> 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 <u>only</u> 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 <u>not</u> 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) <u>and</u> 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 <u>not</u> 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, <u>if</u> they appear in the INTERFACE part, the ordering of a USES declaration may <u>be</u> 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 VIII.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, <u>both</u> must be recompiled.

Unlike former versions of the System, 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 (see Section VI.3).

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 11.3).

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 VII, and in the next section.

VlIl.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, or BASIC) 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 <u>not</u> 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.

VIll.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 the 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.TEXT.   The Linker then asks for the names of library files in which 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? FlX.8<return>
No file FIX.8.CODE
Type <sp>(continue), <esc>(terminate)
Lib file? F1X.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 reads all the necessary routines from the designated codefiles.   It then asks for a destination 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 <u>same</u> 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 <u>Installation Guide.</u>

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.

VIII.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 not 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.

VIII.5.1   Using LIBRARY

When LIBRARY is executed, a prompt asks for an output filename.   The filename must end in .CODE.   LIBRARY will remove 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:

ibrary: N(ew, 0-9(slot-to-slot, E(very, S(elect, C(omp-unit, F(ill,?

```
nput file? SCREENOPS<return>
 0 u  SCREENOP   921              8
 1 s  SEGSCIN1   416              9
 2                               10
 3                               11
 4                               12
 5                               13
 6                               14
 7                               15

'rite 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: units, programs, segment routines, and assembled 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 <u>not</u> 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.

352

IX. 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.

IX.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( 1, J ) );
            start( RED( 3, 4 ), PID );
            start( RED( 5, 5 ), PID, 300 );
            start( RED( J, 1 ), PID, 1+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.

See START in Section V1.2.26 for more details.

IX.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.   They are described fully in Section VI.2.

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.

IX.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.

IX.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.
```

IX.5 Interrupt Handling

A primary goal in providing this collection of concurrency primitives has been to make possible the handling of hardware interrupts with Pascal procedures. A simple example is shown below which attaches a semaphore to a line clock interrupt vector and sounds the bell once every 60 ticks.

```
PROGRAM TOCK;
  const         clockvector = 64;          (* PDP-11 *)
  var           tick : semaphore;
                lpid : processid;

  PROCESS TIMER;
    const       bell = 7;
    var         ticks : integer;
    begin
      ticks := 0;
      repeat
        wait( tick );
        ticks := ticks+1;
        if ticks >= 60 then
          begin
            ticks := 0;
            write( chr( bell ) )
          end
      until false
    end (* TIMER *) ;

begin (* TOCK *)
  set_priority( 100 );
  seminit( tick, 0 );
  attach( tick, clockvector );
  start( timer, lpid, 32, 200 ); {32 word stack, priority=200}
  repeat until false
end.
```

IX.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.

X.   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 <u>Installation
Guide</u>.

X.1   Preparing Assembly Codefiles For Uses Outside Of The System

The utility program COMPRESSOR 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.

COMPRESSOR 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.

X.1.1   Preparing Codefiles For Compression

The assembly procedure(s) must be assembled with the UCSD Adaptable Assembler, and linked with the Linker (see Chapter VII, and Section VIII.4).   Codefiles containing anything other than one segment of linked assembly code will cause COMPRESSOR 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 VII.2.3 and VII.2.8).   .ABSOLUTE'd codefiles can be compressed, but the code produced will be incorrect.

.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 COMPRESSOR to abort.

X.1.2 Running COMPRESSOR

The codefile name is COMPRESSOR.CODE. At the command level, eX(ecute COMPRESSOR. It will respond with the following prompt:

>Assembly Code File Compressor <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 (COMPRESSOR 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, COMPRESSOR will print an error message and abort.

In all the previous prompts, typing the character '!' causes COMPRESSOR to abort.

After receiving information from the prompts, COMPRESSOR reads the entire source file, compresses the procedures, and writes out the entire destination file. Large codefiles may cause COMPRESSOR to abort, if the system does not have sufficient memory space.

While running, COMPRESSOR 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 COMPRESSOR is a file of pure code, which must be loaded and executed directly by user software.

X.1.3   Action and Output Specification

COMPRESSOR 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.

366

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)). COMPRESSOR performs no internal checking for this case.

X.2   Patch

PATCH is a utility which allows hands-on viewing and altering of files.   PATCH is meant for bit-diddling and other such 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.

X.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, 1(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)

X.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, <u>do not</u> 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 ASCll
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 ASCll 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.

X.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.

X.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.)

X.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	1V.0	M_PSEUDO	2	PROG_SEG	R	1	10	4
1	ADD1	1	14	1V.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):     LDC1       400            819001
    11(00B):     EQU1                      B0
    12(00C):     FJP          4            D4F6
    14(00E):     CXG          4      2     940402
    17(011):     RPU          0            9600

DATA POOL:    SEGMENT=ADD1        PROCEDURE=  1   BLOCK=  1    BLOCK OFFSET=   0

   0:000d.   | 0000.  | 4441.DA| 4944.1D| 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):     AD1                       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. Other possible values for M_TYPE are: M_6809, M_PDP, M_8080, M_Z_80, M_GA_440, M_6502, M_6800, and M_9900.

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,

380

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 'I' 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

X.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.

X.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 (4 or 5).   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.

X.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.

X.5   XREF -- The Procedural Cross-Referencer

X.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.

X.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.

X.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.

X.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 '*'.

X.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.

X.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.

X.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.

X.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.

X.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
  Users' 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.

X.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.

X.6.1   The Debugger

This section describes the Debugger utility.   The Debugger can be used as an aid to debugging compiled programs.   lt can be invoked from the main System promptline, or during the exeution of a program (when a breakpoint is encountered).   Memory may be displayed and altered, P-code may be single-stepped, Markstack chains may be displayed and traversed, and so forth.

There are no promptlines explaining the Debugger commands because such prompts would detract from the information displayed by the Debugger itself.   (The commands are detailed in the sections below.)   When a command is entered, there are usually several prompts which may ask for further information such as a segment name, variable offset etc.   lf a space is typed for any of these, the command is exited.   The exception to this is the Breakpoint command as described below.   lf an inappropriate response is given to any prompt (such as ´proc Num?´ 3000) then the response will not be accepted.

ln order to properly use the Debugger, it is necessary to be familiar with the UCSD P-machine architecture. The user should understand the P-code operators, Stack usage, variable and parameter allocation etc.   These topics are discussed in the <u>Internal Architecture Guide</u>.   lt is possible to cause the System to die if the Debugger is used incorrectly.   Also, in order to use the Debugger, it is necessary to have a compiled listing of the program being debugged.   This listing is useful in determining P-code offsets etc., and should be current.

X.6.2   Invoking and Exiting the Debugger

The Debugger may be entered from the main System promptline by typing ´D´. Whenever the debugger is entered in a ´fresh´ state, the prompt, ´DEBUG [version #]´, will appear and a ´(´ will be displayed on the second line.   lf the Debugger is entered in a ´non-fresh´ state, only the ´(´ will appear.   Being in a ´fresh´ state means that the Debugger was not previously active and no breakpoints are currently enabled.

The Debugger may be exited by typing Q(uit, R(esume or S(tep.   lf the Debugger is exited using the Q(uit option, it will be disabled.   lf it is later re-invoked, it will be in a ´fresh´ state.   lf the Debugger is exited using the R(esume option, execution will continue from where it left off and the Debugger will still be active.   lf it is then re-invoked, it will be in a ´non-fresh´ state.   lf the Debugger is exited by using the S(tep option, a single P-code operator will be executed and then the Debugger will be re-invoked (in a ´non-fresh´ state).

Breakpoints are handled by typing B(reakpoint.   After ´B´ is typed, one of the following characters must be typed: S(et, R(emove or L(ist.   (**Note:** There are several two-character commands like this which are used in the Debugger.   lf,

after typing the first character, it is decided to exit from that command, simply type <space> and the main mode of the Debugger will be re-invoked.) If 'S' is typed (after the 'B') then a breakpoint may be set. The user may have, at most, five breakpoints numbered 0 through 4. The first prompt is 'Set Break #?': a digit 0..4 should be typed followed by <space>. The next prompt is 'Segname?': the name of the desired segment should be typed followed by <space>. Then 'Proc #?' appears: the number of the desired procedure should be typed followed by <space>. Then 'Offset #?' appears: the desired offset within the procedure should be typed followed by <space>. A breakpoint is then set and if, during execution resumption, that segment, procedure and offset are encountered, the Debugger will be automatically re-invoked.

When setting a breakpoint, a space may be typed for the break number, segment name etc. Rather than exiting the breakpoint command (as would happen with other commands), the previous breakpoint's information will be used. For example, if it is desired to break in the same segment and procedure but with a different offset, a space may be typed for everything except the offset.

If, after typing B(reakpoint, an 'R' is typed, a breakpoint may be removed. The prompt 'Remove break #?' appears. The number of the breakpoint, 0..4, should be typed followed by <space>: the indicated breakpoint is removed.

If after typing B(reakpoint, an 'L' is typed, the current breakpoints are listed.

The Debugger may be memlocked or memswapped (see the descriptions of those intrinsics) by using the M(emory command at the outer level. ML will memlock and MS will memswap the Debugger.

X.6.3   Displaying and Altering Memory

By typing V(ar, data segment memory may be displayed. This is another two-character command and may be followed by G(lobal, L(ocal, I(ntermediate or P(rocedure. If 'G' or 'L' is typed, the prompt 'Offset #?' appears: the desired offset into the data segment should be typed. If 'I' is typed, the 'Delta Lex Level?' is also prompted. If 'P' is typed, an offset within a specified procedure may be displayed: 'Segment name?', 'Procedure #?' and 'Offset #?' are all prompted in sequence.

When any of these options are used, a line similar to the following is displayed:

(g) S=KERNEL   P#1   VO#1 2C1A: 0B 05 53 43 41 4C 43 61   -SCALCa

... in this example, a Global (g) segment of memory is displayed. The segment is KERNEL, procedure 1, variable offset 1 at absolute hex location 2C1A. Following this, eight bytes are displayed, first in HEX and then in ASCII (a '-' indicates

that the character is not a printable ASCll character).

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 following three options are available: U(p, D(own and L(ist. If L(ist is invoked, all of the currently existing mark stacks will be displayed, with the most recently created one first. An entry in the list will resemble the following:

(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA msenv=FEE8

If the U(p or D(own options are used, the frame of reference moves up or down one link and variable listings (using the 'V' command) change accordingly.

After a line has been displayed by the V(ar command, a '+' or '-' may be typed. This displays the succeeding or preceding eight bytes of memory. If a '/' is typed, then the line displayed above it may be altered in hex mode. If a ' ' is typed, then the line displayed above it may be altered in ASCll mode. When altering in hex mode, any characters which are to be left unchanged may be skipped by typing <space>. In the ASCll mode, any characters to be left unchanged may be skipped by typing <return>.

X.6.4 Further Single-Stepping Options

When the single-stepping mode (described in Section X.6.2 above) is used, one P-code operator is executed at a time. When control is returned to the Debugger, it displays various pieces of information if they are desired. In order to select what will be displayed, the E(nable mode should be used. After typing 'E', the following options are available: R(egister, P(code, M(arkstack, A(ddress and L(oad. Any or all of these options may be enabled at the same time.

If R(egister is enabled, a line such as the following will be displayed after each single step:

(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 will be displayed after each step:

(cd) S=HEAPOPS P#3 O#23   LLA 1

If M(arkstack is enabled, a line such as the following will be displayed after each step:

(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA msenv=FEE8

If A(ddress is enabled, a line such as the following will be displayed after each step:

(a) S=HEAPOPS P#3 O#23 2C1A: 0B 05 53 43 41 4C 43 61 -SCALCa

In order to initialize this address to a given value, there is an A(ddress mode at the outer level. When 'A' is typed, 'Address ?' appears. An absolute address, in hex, should be typed in. At this point, eight bytes are displayed starting at that address. Also, that address is now displayed if the E(nable A(ddress option is on.

Enabling E(very will cause 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.

Also, at the outer level, there is a P(code option. This option asks for 'Segment name?', 'Proc #?', 'Start Offset #?' and 'End Offset #?' and disassembles the indicated portion of code. This may be useful during single-step mode if it is desired 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.

X.6.5 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    Proc #? 1    Offset #? 6
(EP)
(R)
```

After setting the break point 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 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 that sp points to.
What we discover is 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.

X.6.6   Summary of the Commands

D(ebug   Enters Debugger from main promptline

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

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

V(ariable
  G(lobal   Displays global memory
  L(ocal   Displays local memory
  I(nter   Displays intermediate memory
  P(roc   Displays data segement of given proceudre
  E(xtended   Displays varialbes in another segment

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

E(nable   Enables the following to be displayed during single step
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

A(dress   Displays a given address

P(code   Dissassembles a given procedure

M(emory
  L(ock   Memlocks the Debugger
  S(wap   Memswaps the Debugger

X.7 The RECOVER Utility

RECOVER is a utility which attempts to re-create the directory of a disk whose directory has accidentally been destroyed. It is shipped as RECOVER.G.CODE (or RECOVER.T.CODE for Terak users).

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 1V.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..12] 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.

XI.   APPENDICES

XI.A   Appendix A -- Execution Errors

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 I(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

See also the information in Section VI.2.14, IORESULT.

XI.C Appendix C -- Device Numbers

```
Device Number        Volume Name
        0            <for System use>
        1                CONSOLE:
        2                SYSTERM:
        3                GRAPHIC:
        4            <System disk '*'>
        5            <other disk>
        6                PRINTER:
        7                REMIN:
        8                REMOUT:
        9
       10            <user-defined disks
       11              or other devices>
       12
       ...
```

Devices with numbers 9..12 or greater are user-defined devices. Devices 4 and 5 are usually floppies, though they may be other sorts of block-structured devices. Devices 1..3 are described in Chapter lll -- Files and File-handling. REMIN: and REMOUT: are often set to the same bidirectional port.

More information on devices may also be found in the Installation Guide.

XI.D   Appendix D -- Reserved Words

Standard Pascal Reserved Words

and	set
array	
	then
begin	to
	type
case	
const	until
div	var
do	
downto	while
	with
else	
end	

UCSD Pascal Reserved Words

file
for   external
forward
function   interface
                   implementation
goto
                   process
if
in   segment

label

mod

nil
not

of
or

packed
procedure
program

record
repeat

Standard Predeclared Identifiers
----

abs	read
arctan	readln
atan	real
	reset
Boolean	rewrite
	round
char	
chr	sin
cos	sqr
	sqrt
eof	succ
eoln	
exp	text
	true
false	trunc
get	[unpack]
input	write
integer	writeln
ln	
maxint	
new	
odd	
ord	
output	
[pack]	
page	
pred	
put	

UCSD Predeclared Identifiers

attach	pos
	processid
blockread	pwroften
blockwrite	
	release
close	
concat	scan
copy	seek
	semaphore
delete	seminit
	signal
exit	sizeof
	start
fillchar	str
	string
gotoxy	
	time
halt	
	unitbusy
insert	unitclear
interactive	unitread
ioresult	unitstatus
	unitwait
keyboard	unitwrite
length	varavail
	vardispose
mark	varnew
memavail	
memlock	wait
memswap	
moveleft	
moveright	

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 garbage on line
6: Input line over 80 characters
7: Not enough ifs
8: Must be declared in ASECT before use
9: Identifier previously declared
10: Improper format
11: EQU expected
12: Must EQU before use if not to a label
13: Macro identifier expected
14: Word addressed machine
15: Backward ORG not allowed
16: Indentifier expected
17: Constant expected
18: Invalid structure
19: Extra special symbol
20: Branch too far
21: Variable not PC relative
22: Illegal macro parameter index
23: Not enough macro parameters
24: Operand not absolute
25: Illegal use of special symbols
26: Ill-formed expression
27: Not enough operands
28: Cannot handle this relative
29: Constant overflow
30: Illegal decimal constant
31: Illegal octal constant
32: Illegal binary constant
33: Invalid key word
34: Unexpected end of input - after macro
35: Include files must not be nested
36: Unexpected end of input
37: Bad place for an include file
38: Only labels & comments may occupy column one
39: Expected local label
40: Local label stack overflow
41: String constant must be on 1 line
42: String constant exceeds 80 chars
43: Illegal use of macro parameter

44: No local labels in ASECT
45: Expected key word
46: String expected
47: Bad block, parity error (CRC)
48: Bad unit number
49: Bad mode, illegal operation
50: Undefined hardware error
51: Lost unit, no longer on-line
52: Lost file, no longer in directory
53: Bad title, illegal file name
54: No room, insufficient space
55: No unit, no such volumn on-line
56: No file, no such file on volume
57: Duplicate file
58: Not closed, attempt to open an open file
59: Not open, attempt to access a closed file
60: Bad format, error in reading real or integer
61: Nested macro definitions not allowed
62: '=' or '<>' expected
63: May not EQU to undefined labels
64: Must declare .ABSOLUTE before first .PROC

Z-80 based machines
Specific error messages:

76: Incorrect operand format
77: Close paren ")" expected
78: Comma "," expected
79: Plus "+" expected
80: Open paren "(" expected
81: Stack pointer "SP" expected
82: "HL" expected
83: Illegal "CC" condition code
84: Register "C" expected
85: Register "R" expected
86: Register "A" expected

PDP-11 based machines
Specific error messages:

76: Closing paren ")" expected
77: Register expected
78: Too many special symbols
79: Unrecognizable operand
80: Register reference only
81: First operand must be a register
82: Comma expected
83: Unimplimented instruction
84: Must branch backwards to label

8080 based machines have no specific error messages.

6502 based machines
Specific error messages:

76: Index register required
77: "X" or "Y" expected
78: Zero-page address required
79: Illegal use of register
80: Index register expected
81: Ill-formed operand
82: "X" expected for indexed addressing
83: Must use "X" register

6800 based machines
Specific error messages:

76: "X" expected for indexed addressing
77: "A" or "B" expected

9900 based machines
Specific error messages:

76: Illegal immediate operand
77: Index must be WR
78: Close paren ")" expected
79: Indirect and autoincr must be WR
80: Autoincr must be WR indirect
81: Comma "," expected
82: No operand allowed
83: Illegal map file

Z8 based machines
Specific error messages:

76: Too many symbols
77: Operand expected
78: Bad data value
79: ')' expected
80: Bad operand type
81: Odd register
82: Unknown instruction
83: Working register expected
84: Indirect or register expected
85: Condition code expected

XI.F   Appendix F -- Summary of Differences between UCSD Pascal and Standard Pascal

1.   String Handling

STRING is an intrinsic data type, consisting of a PACKED ARRAY OF CHAR together with a length.   Strings may be assigned, passed, and input or output.

The following UCSD intrinsics are for the manipulation of strings:

function CONCAT (source [, source]... : string]): string

function COPY (source: string; index, size: integer): string

procedure DELETE (destination: string; index, size: integer)

procedure INSERT (source, destination: string; size: integer)

function LENGTH (source: string): integer

function POS (pattern, source: string): integer

2. I/O Intrinsics

READ, READLN and WRITE, WRITELN may only be used on files of type TEXT (= FILE OF CHAR).

In addition to the standard file types, files may be untyped or INTERACTIVE. The predefined files INPUT, OUTPUT, and KEYBOARD, are INTERACTIVE in UCSD Pascal.

KEYBOARD is a non-echoing equivalent of INPUT.

If a file is INTERACTIVE ...

    the EOF function is set by input of an <etx> character;
       <etx> is defined in SYSTEM.MISCINFO;

    the EOLN function is set by a <return>;

    READ and READLN will perform a GET before loading the
    file's window variable; the effect of this is to require
    that a READ or READLN be done on an INTERACTIVE file
    before testing EOF or EOLN;

    RESET does not load the file's window variable.

If a file is untyped ...

    all I/O to that file must use the BLOCKREAD and BLOCKWRITE
    intrinsics.

RESET and REWRITE generally behave as standard intrinsics, but they both may take an optional second parameter that is a disk filename -- this makes the Pascal file equivalent to the physical disk file.

The intrinsic SEEK does random access on files. The intrinsic CLOSE controls the fate of a disk file. UNITREAD, UNITWRITE, and other UNITxxx intrinsics are for direct control of peripheral devices. IORESULT returns the status of an I/O operation.

WRITE and WRITELN are incapable of writing Booleans or record variables. STRINGs and PACKED ARRAYs OF CHAR may be output in a single WRITE.

These are the UCSD intrinsics that handle devices and files:

```
function BLOCKREAD ( fileid: {untyped} file;
                           buffer: packed array of char;
                           blocks [, relblock]: integer ): integer

function BLOCKWRITE ( fileid: {untyped} file;
                           buffer: packed array of char;
                           blocks [, relblock]: integer ): integer

procedure CLOSE ( fileid: {any sort of} file; <option> )
     <option> ::= , LOCK | , NORMAL | , PURGE | , CRUNCH

function IORESULT : integer

procedure SEEK ( fileid: {any sort of} file, recnum: integer )

function UNITBUSY ( unitnumber: integer ): Boolean

procedure UNITCLEAR ( unitnumber: integer )

procedure UNITREAD ( unitnumber: integer;
                           buffer: packed array of char;
                           length
                           [, [blocknumber] [, option]]: integer )

procedure UNITWAIT ( unitnumber: integer )

procedure UNITWRITE ( unitnumber: integer;
                           buffer: packed array of char;
                           length
                           [, [blocknumber] [, option]]: integer )
```

3. Memory Management

A SEGMENT PROCEDURE behaves as any other procedure, but is disk-resident and present in main memory only when it is being executed.

A UNIT is a separately compiled collection of procedures and data structures. This is an outline of a UNIT:

    UNIT <unitname>;

    INTERFACE
        {declarations and procedure headings appear here}
        {these and only these may be used by the host}

    IMPLEMENTATION
        {declarations and procedure code appear here}
        {this portion is private to the UNIT}

    BEGIN
        {initialization code}
        ***;
        {termination code}
    END

The initialization code is executed before any host program code. The host program invokes a unit by:

    PROGRAM <program_name>;
        USES <unitname>, <more_unitnames ... >;

The standard intrinsics NEW and DISPOSE are implemented.

The following UCSD intrinsics are meant for memory management:

    procedure MARK (var heapptr: ^integer)

    function MEMAVAIL : integer

    procedure MEMLOCK (seglist: string)

    procedure MEMSWAP (seglist: string)

    procedure RELEASE (var heapptr: ^integer)

function VARAVAIL (seglist: string): integer

procedure VARDISPOSE (pointer: ^{any type}; count: integer)

procedure VARNEW (pointer: ^{any type}; count: integer)

4. Concurrency

A PROCESS is declared as a procedure, and may be STARTed any number of times by the main program. Processes may be controlled by semaphores. The UCSD predeclared type SEMAPHORE is a subrange: [0..maxint]. The UCSD predeclared type PROCESSID is used only by the System.

Example: PROCESS ZIP;
          BEGIN ... END;

        process DINNER (var SPLIT, BLACKEYED: peas);
          begin ... end;

The following UCSD intrinsics are for the control of processes:

    procedure ATTACH (sem: semaphore; vector: integer)

    procedure SEMINIT (var sem: semaphore; sem_count: integer)

    procedure SIGNAL (var sem: semaphore)

    procedure START (<process call>;
                     [, id: processid;]
                     [, stacksize: integer;]
                     [, priority: byterange])
      <process call> ::= {a normal procedure call}
      type byterange: 0..255

    procedure WAIT (var sem: semaphore)

5. Miscellaneous

Syntax variations:

CASE statements fall through if no label matches the selector.

Comments may be enclosed by either '{ }' or '(* *)';
the two _different_ types may be nested (only one comment deep).

'=' and '<>' may be used. for extended array or record
comparisons.

GOTOs are restricted to labels within the same block.

procedure EXIT (procid: <procedure identifier>)
... is used to immediately abort a procedure.

A length attribute defines a LONG INTEGER,
e.g.,
        var LONG: integer[8];

... the length defines the _minimum_ number of digits
in the integer

procedure STR (value: integer[n]; destination: string)
... converts an integer into a string; usually used for the
output of long integers. (The length attribute is optional.)

PACK and UNPACK are not implemented. Packing and unpacking
are done automatically. A PACKED ARRAY OF CHAR may be
assigned, input, and output, as a single entity
(as with a STRING).

Packed variables may not be used as call-by-reference (_var_)
parameters.

Sets of subranges of integers must include only positive
integers.

Set comparisons must be between sets of the same underlying type.

The arctangent function may be called either ATAN or ARCTAN.

The following UCSD intrinsics are for the handling of large arrays:

procedure FILLCHAR (destination: packed array of char;
          length: integer; character: char)

procedure MOVELEFT (source, destination {any sort of} array;
          length: integer)

procedure MOVERIGHT (source, destination {any sort of} array;
          length: integer)

function SCAN (length: integer;
          <partial expression>; <partial expression>;
          source: packed array of char): integer
          <partial expression> ::= '<char>' | <>'<char>'

function SIZEOF ({any variable or type identifier}): integer

The following are miscellaneous UCSD intrinsics:

procedure GOTOXY (x, y: integer)

procedure HALT

function PWROFTEN (exponent: integer): real

procedure TIME (var hiword, loword: integer)

6. Writing a Transportable Program

These are a few hints and suggestions.

A.   Avoid the peculiarities of UCSD Pascal detailed above.

B.   Untagged case variant records often cause trouble.
     The value of the case tag will be checked only by
     the runtime system, or not at all.

C.   Assume nothing about variable allocation -- the size of
     variables, packing algorithms, representations of real
     numbers and of Booleans all vary from system to system.

D.   Make sure variables are unique in the first 8 characters.

E.   Don't assume that all of an expression will be evaluated.
     Some compilers try to optimize around subexpressions.

XI.G Appendix G -- Summary of Differences Between Versions

The UCSD p-System (the latest of its names) has gone through a number of incarnations since its first release to the public. The names it has borne are: 1.3, 1.4, 1.5, 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. The latest 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.

IV.0 offers upward compatability at the source code level, introduces multitasking to interpreter-based implementations of UCSD Pascal, and provides more flexible and cleaner memory-management techniques than previous versions.

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, 1.3 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, 1.3's System editor was YALOE. 1.3 ran on PDP-11's and LSI-11's.

1.4 was the first version to be available on other microprocessors, including 8080's and Z80's running CP/M. 1.4 also introduced the full Screen Oriented Editor.

1.5 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 1.5. 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 is new, and pulls together the user-level features of the last three versions.

A.   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.0:

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.

XI.H   Appendix H --   Converting Pascal Programs to IV.0

1.   Introduction

This section describes changes that must be made to Pascal and assembly language programs in order to run them on the IV.0 System.   Some of the changes are concerned with interfaces to the System; others affect various and sundry Version II and III programming practices.

2.   Converting Pascal Programs

2.1   Use (And Misuse) Of The Heap

Version IV is the first version of the UCSD p-System to implement a true heap, as defined in standard Pascal.   For this reason, most of the venerable and hence treasured programming tricks associated with the rudimentary heap implementations of past versions must be laid to rest.

Consecutive calls to standard procedure NEW no longer guarantee the allocation of a contiguous area of memory; therefore, the practice of creating variable-sized buffers using a sequence of NEW's will not work.   The UCSD intrinsics VARNEW and VARDISPOSE should now be used to allocate variable-sized buffers.   The Version IV heap is as sensitive to range violations as the stack has always been -- use it with corresponding care.

The standard procedures MARK and RELEASE must be used only for the purposes for which they were devised.   Using a MARK'ed pointer as a pointer to heap data does not work in Version IV.   The contents of a MARK'ed pointer must not be altered in any way until the matching call to RELEASE has been performed. RELEASE's must only be performed on variables that have been previously MARK'ed (and not yet RELEASE'd).

2.2   Code Segment Management

With the code pool scheme, code segments need to be loaded from disk much more frequently (and less predictably) than in the past.   Several System segments may require loading during the course of a single System call: the System disk must be on-line to complete the call.   This can affect the usefulness of programs which manipulate the disk volumes (such as the System Filer).

Two solutions address this problem.   A program can use the memory management procedure MEMLOCK to lock into the code pool all code segments required for its execution.   The procedure MEMSWAP can later be used to unlock these segments

(segments should not be left locked if they do not need to be; this uses much space and can cramp the Operating System).

The other solution is more direct (but possibly less efficient). If direct control of code residency is undesirable, the System will prompt the user to place the proper disk in a drive so the required code segment may be loaded.

2.3 Compiler Directives

The F (byte-flipping), G (no goto's), and S (segment swapping) compiler directives have no effect in Version IV and may be removed. Version IV P-code is byte-asexual, so the F option is now irrelevant. Goto restrictions were a carry-over from the university and are no longer needed (indeed, they conflict with current Pascal standards). User-controlled segment swapping is no longer necessary; the Compiler now handles swapping automatically.

Leaving these directives in your source code causes no harm at present. However, it is not impossible that in the future these letters will acquire new meanings as compiler directives, so the most prudent course is to remove them from your programs.

2.4 Compiling System-Level Programs

Examples pertaining to the following discussion appear at the end of this section.

The outermost (Operating System) lexical level common to versions II and III no longer exists. The user program directive U- sets the compiler options R- and I-, and allows units to be compiled with reserved System names (see the section below for details on Version IV units); however, it does not affect the lexical level of programs or units. This change has the following effects on existing System-level programs:

> The outermost dummy lex level is invalid and must be removed. There being no distinction between a System and a user program, the segment procedure declaration for the System program in question must be replaced with a normal program declaration. The dummy parameters associated with the segment declaration are no longer necessary. Also, the dummy body at the end of existing System programs which corresponds to the old System lexical level must be removed.

> Dummy segment procedure declarations are unnecessary and may be removed, as Version IV segment numbers are not System-wide resources; their scope only extends to the enclosing program or unit. Failure to remove the dummy declarations will not affect the execution of a program,

but will cause an unnecessary increase in the size of its codefile.

The Version IV System globals reside in the interface section of the Operating System's KERNEL unit. System-level programs which include the file GLOBALS.TEXT must now use unit KERNEL. The version of the kernel unit contained in the standard SYSTEM.PASCAL does not contain an interface section, so a separate codefile containing the unit with its interface section is supplied.

The System-level variables and data type declarations in the kernel unit are almost identical to those of the older System globals. The only objects missing in Version IV are the variable DEBUGINFO in the System variables and the BUGSTATE and SEGTABLE fields in SYSCOM (the reasons being that the Version IV debugger doesn't need the debug fields, while the segment environment is handled in a very different manner). All other variables and data types have the same identifier names.

Programs which use modified versions of GLOBALS.TEXT to access a subset of the old System globals can do so in Version IV by moving their own global declarations into a stubbed version of the kernel unit's interface section. This is done by declaring a kernel unit containing the appropriate declarations in its interface section and using it in the manner described below. This dummy kernel unit must be compiled with the U- option, and the unit name must be KERNEL. Care should be taken to ensure that the subset declarations correspond with the Version IV System globals.

Programs which require direct (as opposed to compiler-generated) accesses to Operating System procedures must explicitly use the Operating System unit containing the needed routines. This is done in a manner similar to use of the kernel unit described below. A description of the Operating System unit names, interfaces, and file names can be found in the Internal Architecture Guide.

Programs which reference the System globals to gain access to the screen control characters and date that reside in SYSCOM will work correctly in Version IV for the time being; however, the data within SYSCOM is currently also contained in the screen control unit (described in Section III.6 of the Installation Guide). The screen control unit will replace SYSCOM in the near future, so it is desirable to make the extra effort now to move user and System programs away from SYSCOM dependencies.

Examples:

Before1

```
(*$U-*)
program System_level;

(*$1 GLOBALS.TEXT*)

segment procedure ll_style(dum1,dum2:integer);

segment procedure dummy2;
begin
end;
...

segment procedure dummy9;
begin
end;

segment procedure mysegment;
begin
...
end;

begin (*ll_style*)
...
mysegment;
...
end;

begin (*dummy outerblock*)
end.
```

After1

In this example, KERNEL.CODE is the file containing the kernel unit's interface section.

```
program IV_style;
uses (*$U KERNEL.CODE*) kernel;

segment procedure mysegment;
begin
...
end;

begin (*IV_style*)
...
mysegment;
...
end.
```

Before2

```
(*$U-*)
program System_level;
type myuserinforec = record
                            stub: integer;
                   end;
var    filler: array   0..6   of integer;
        userinfo: myuserinforec;

segment procedure ll_style(dum1,dum2:integer);

segment procedure dummies2to9;
begin
end;

segment procedure mysegment;
begin
...
end;

begin (*ll_style*)
...
mysegment;
...
end;

begin (*dummy outerblock*)
end.
```

After2

```
(*$U-*)
program lV_style_l;

    unit kernel (*dummy*);
    interface
    type myuserinforec = record
                            stub: integer;
                    end;
    var   filler: array   0..6   of integer;
          userinfo: myuserinforec;
    implementation
    end;

uses kernel;

    segment procedure mysegment;
    begin
    ...
    end;

    begin (*lV_style_l*)
        ...
        mysegment;
        ...
    end.
```

2.4   Architectural Ramifications (Dirty Tricks)

The physical in-memory relationship between parameters and declared variables has changed in Version 1V; therefore, programs which depend on the old architecture must be changed. The following is an example (courtesy of the Version 11 Filer) of this problem:

```
procedure GetAddr (var MyVar: MyType);
    var TrickArray   0..0   of integer;
    AddressOfActualParameter: ^MyType;
begin
    (*$R-*)
    AddressOfActualParameter := TrickArray   -1
    (*$R+*)
end;
```

This procedure could obtain the memory address of a variable of type MyType by making the assumption that local variables are allocated in memory immediately following the procedure's parameters. This assumption is true in version 11, but false in Version 1V. Programs infected with machinations of this ilk will need to be modified.

2.5   Dummy Segment Procedures And The System Librarian

In versions 11 and 111, it is possible to create and maintain programs larger than can be compiled in one shot (due to memory constraints) by compiling each segment of the program separately. The tools used for this task are the System Librarian and a collection of programs, each of which contains only the necessary variable declarations; a single segment procedure, and sufficient dummy segment procedure declarations to assign the correct segment number to the real code segment.

It is assumed that the above description is sufficient to elicit either a nod of recognition from the experienced System user or a squawk of delight from the user who was previously unaware of this practice; in both cases, the party is over! Because the 1V.0 compiler performs on-the-fly assignment of local segment numbers to a program (standard and System procedure calls get local segment numbers), the replacement of a code segment by a dummy body may cause a different segment number to be assigned to the target segment procedure. There exists no simple method for determining the local segment number assigned to most segment procedures in a program containing multiple segment procedures, explicitly used units, and implicitly used Operating System units.

To wit, using LIBRARY to combine the separately compiled segment procedures will not produce executable codefiles. Version 1V presents an elegant solution to

programs which have required this treatment in the past: modularize the program by splitting it into a collection of separately compilable Version IV units.

2.6   Compiling Units

Version IV accepts the syntax for regular, separate, and intrinsic units as input, but maps them all into a single unit scheme. The Operating System unit names are reserved for System use only; this feature is enforced in the Compiler by allowing the compilation of units with reserved names only when the U- Compiler option is used. A list of the reserved unit names may also be found in the Internal Architecture Guide.

Reserved Unit Names

KERNEL	PASCALIO
HEAPOPS	OSUTIL
CONCURRENCY	GOTOXY
STRINGOPS	SCREENOPS
REALOPS	EXTRAHEAP
LONGOPS	EXTRAIO
FILEOPS	DEBUGGER
COMPUNIT	SOFTOPS

Version IV units must contain an interface and an implementation section, even if one or the other is empty.   Intrinsic data units from version II.1 may require the insertion of the reserved word 'implementation' before the 'end' in order to compile successfully.

2.6   Program Headings

Contrary to past versions of the UCSD Pascal Compiler, program or unit headings (i.e., Program stuff; or Unit stuff;) are mandatory. The Compiler will give an error for programs lacking a heading.

2.7   Standard Real-Valued Functions

Version IV does not require the statement 'uses transcendentals' when a program uses standard real-valued functions such as SIN and COS; if present, it must be removed before you compile the program under IV.0. (This is true for all the IV.0 implementations.)

3. Converting Assembly Language Programs

3.1 Macro Parameters and ASCII Strings

Macro parameters are not expanded within ASCII strings in the Version IV Adaptable Assemblers; this is not compatible with previous assemblers. It was necessary to sacrifice the calf of flexibility on the altar of reliability. (P.S. - apologies are due to those who liked and used this feature.)

3.2 Assembler Identifiers

Two changes have occurred to assembly language identifiers in Version IV. First, lower-case alphabetic characters are allowed in identifiers; as in Pascal, they are internally mapped into their upper-case equivalents. Second, the underscore character '_' is no longer significant in identifiers; this too is consistent with Pascal usage.

Example of equivalent assembly language identifiers:

        readloop
        Read_Loop
        READLOOP

3.3 Pascal/Assembly Language Procedure Interface

Byte-array variables (types string and packed array of char) passed as value parameters are handled differently in Version IV. A two-word string descriptor is passed in place of the old one word pointer; its processing will require some extra assembly code.

Consult Chapter VIII for more details.

The order and number of parameter words pushed on the stack prior to an assembly procedure/function call is different for Version IV. The function return words are now beneath all parameters on the stack, rather than being on TOS. Assembly procedures have 0 words of function return space on the stack, real-valued functions have 2 words of return space, and all other functions have 1 word of return space. As in previous versions, these words must be popped from the stack by the assembly routine before the function return value is pushed.

Again, you should consult Chapter VIII.

3.4   Assembly Level Stack Manipulation

Assembly routines which allocate memory above the hardware stack pointer for data space may require changes. In Version IV, the code pool can be as close as 40 words to the hardware top of stack; because assembly routines cannot determine the code pool's location, the routines must use the stack sparingly in order to prevent later System crashes.

3.5   Radix Switch Characters

All versions of the Adaptable Assembler now use the same characters to indicate the radix of a number; thus, source code for some versions may require changes. The two most significant changes are:

Binary integer constants are defined with the radix switch character 'T'.
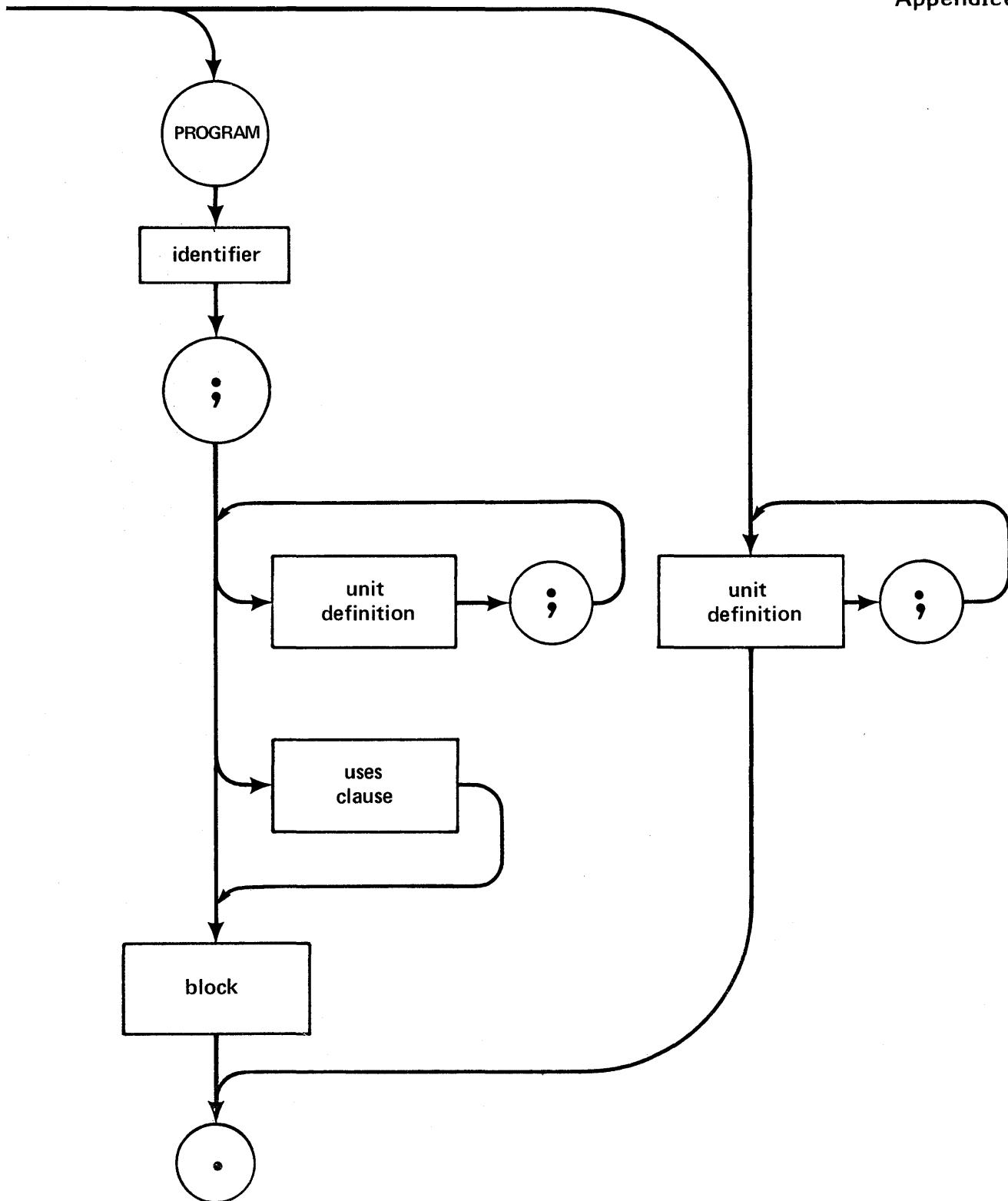
Octal integer constants are defined with the radix switch character'Q'.

Refer to Chapter VIII for more details.
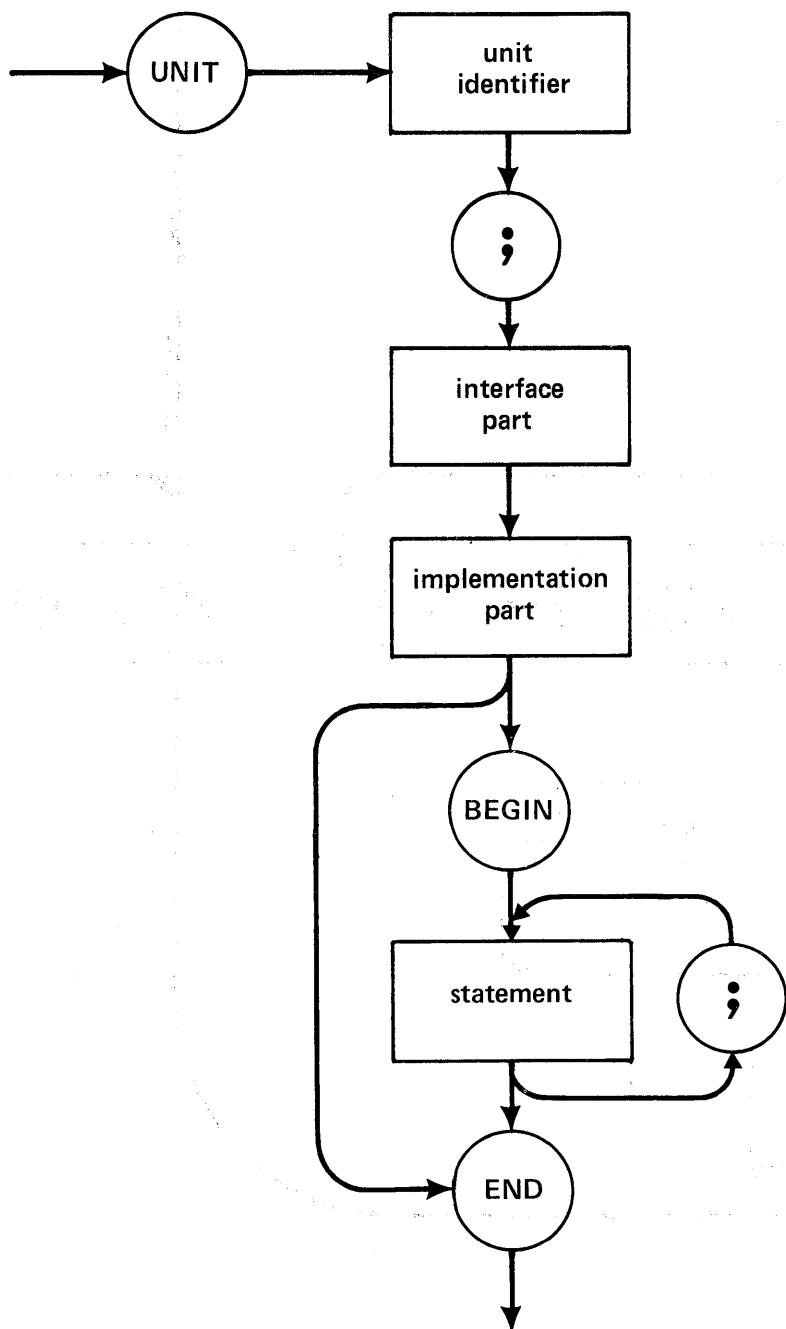
X1.1 Appendix 1 -- Railroad Diagrams for UCSD Pascal Syntax
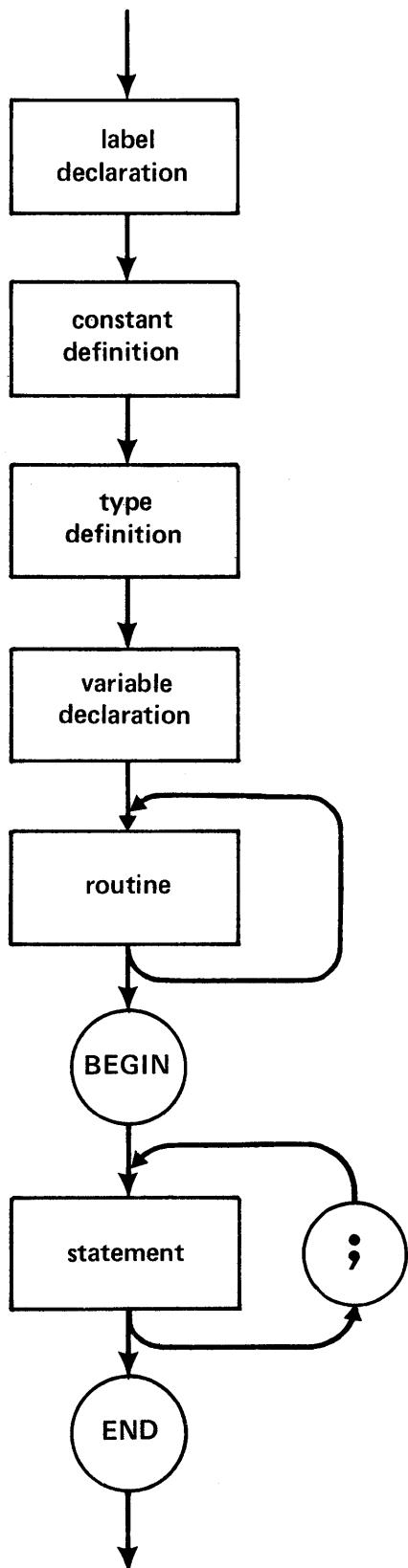
These railroad diagrams are provided as an aid to understanding UCSD Pascal syntax. They show the full syntax, but do <u>not</u> attempt to represent the internal workings of the Compiler.

<unit definition >

⟨uses clause⟩

USES → unit identifier

,

< interface part >

INTERFACE

uses
clause

constant
definition

type
definition

variable
declaration

routine
heading

< implementation part >

IMPLEMENTATION

uses
clause

label
declaration

constant
definition

type
definition

variable
declaration
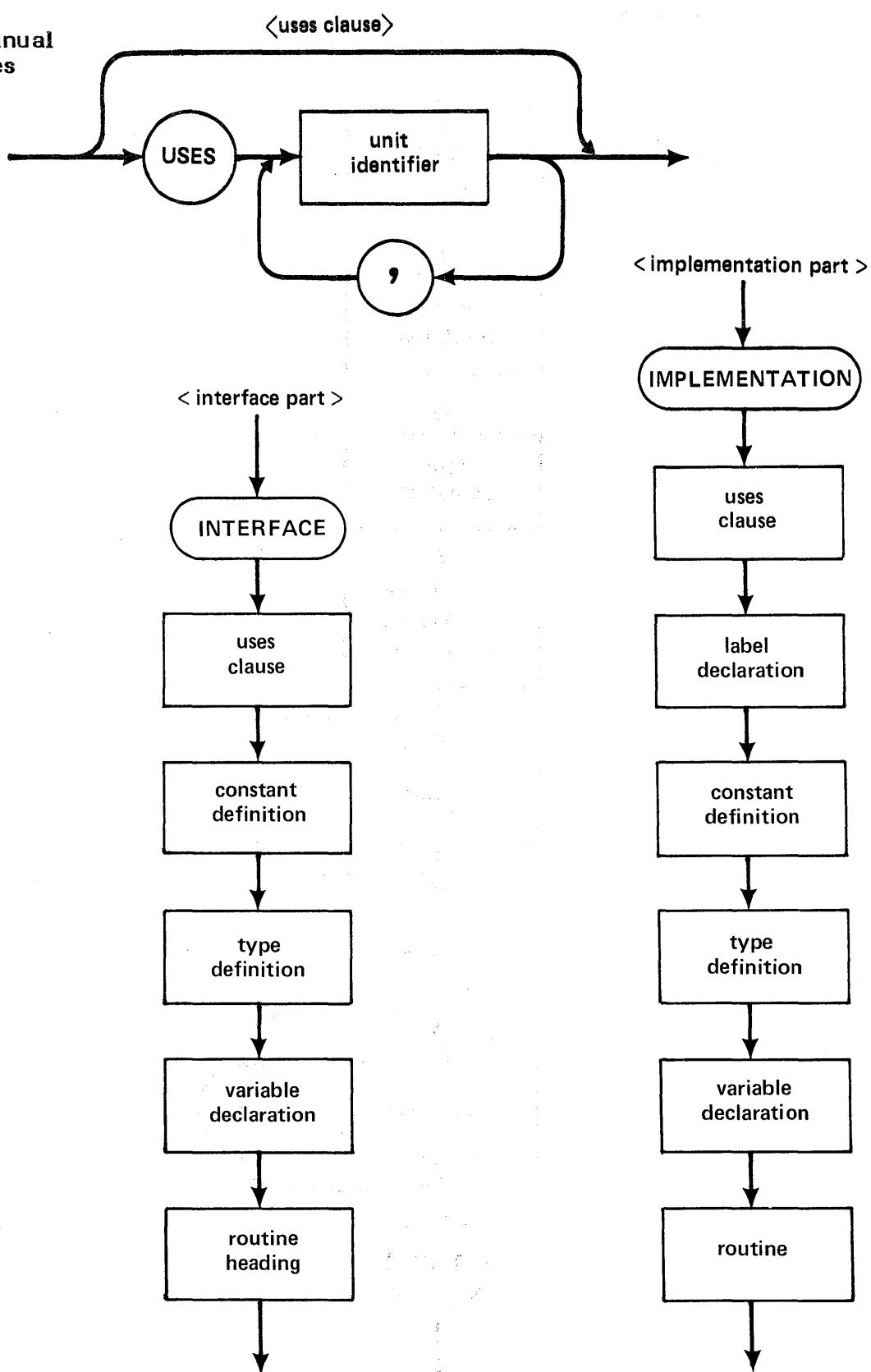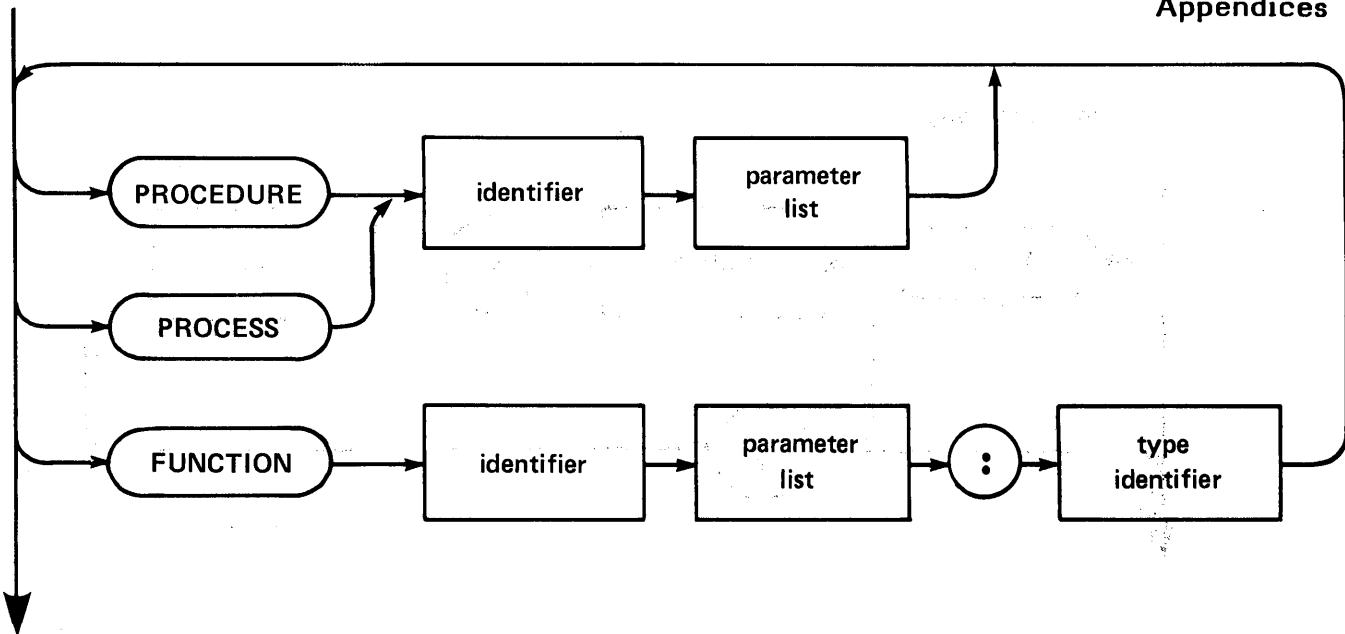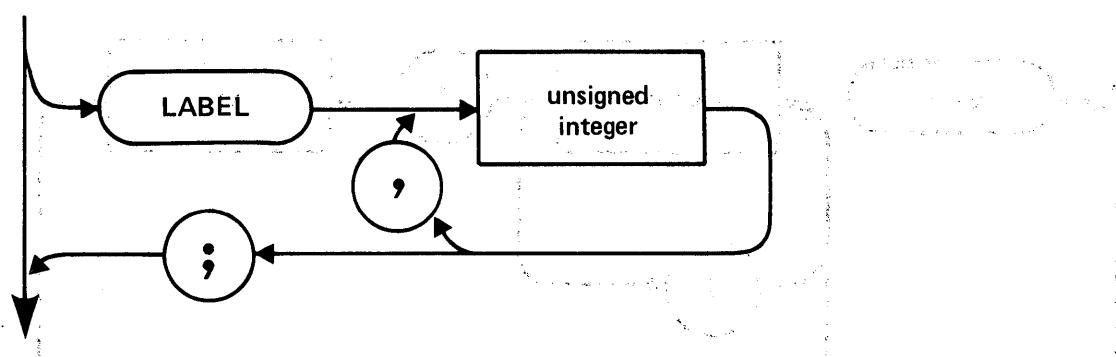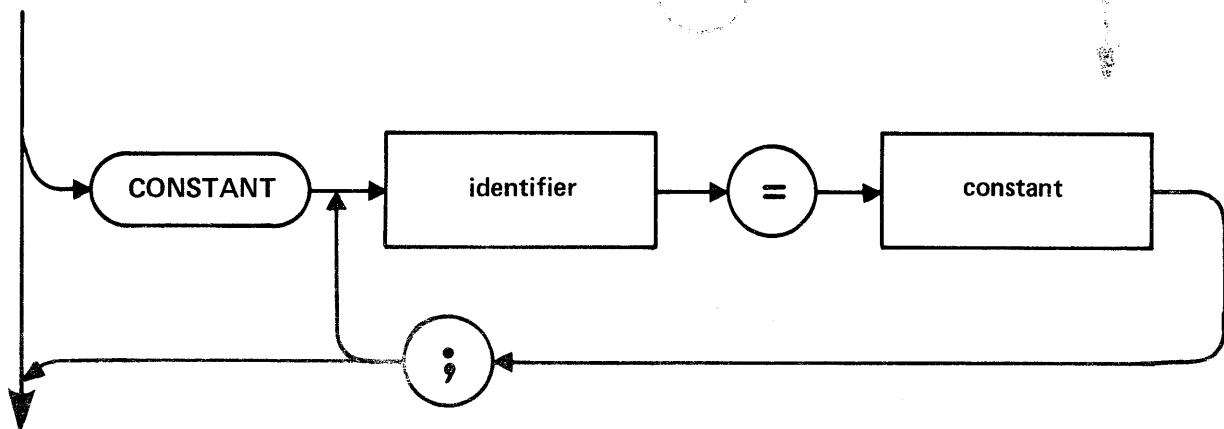
routine

⟨routine heading⟩

⟨label declaration⟩
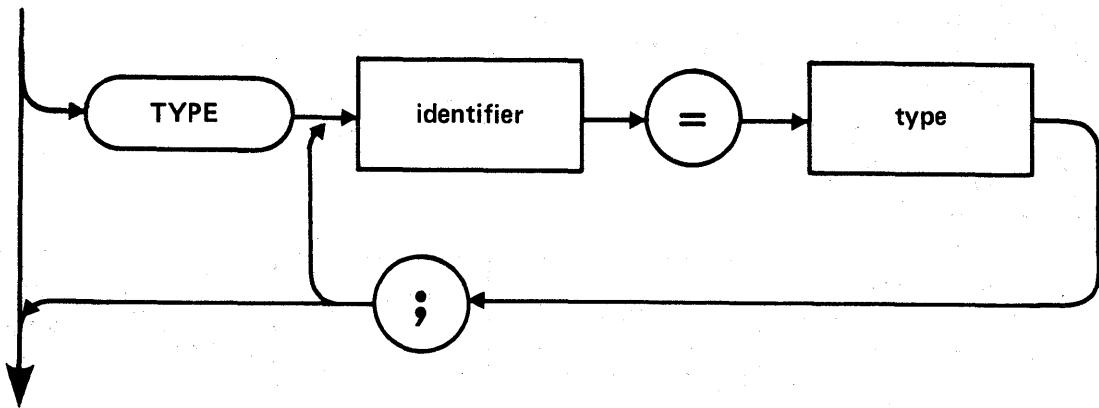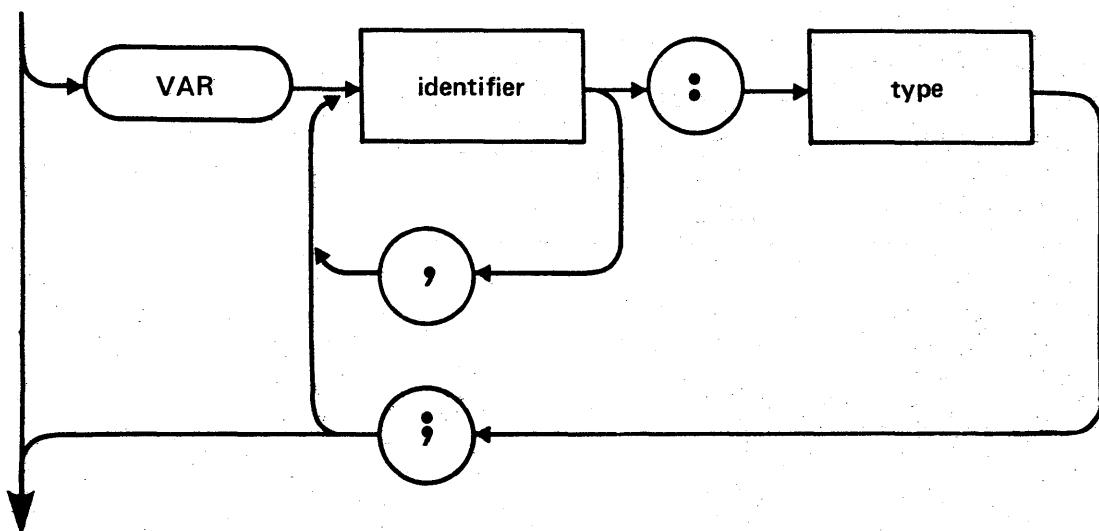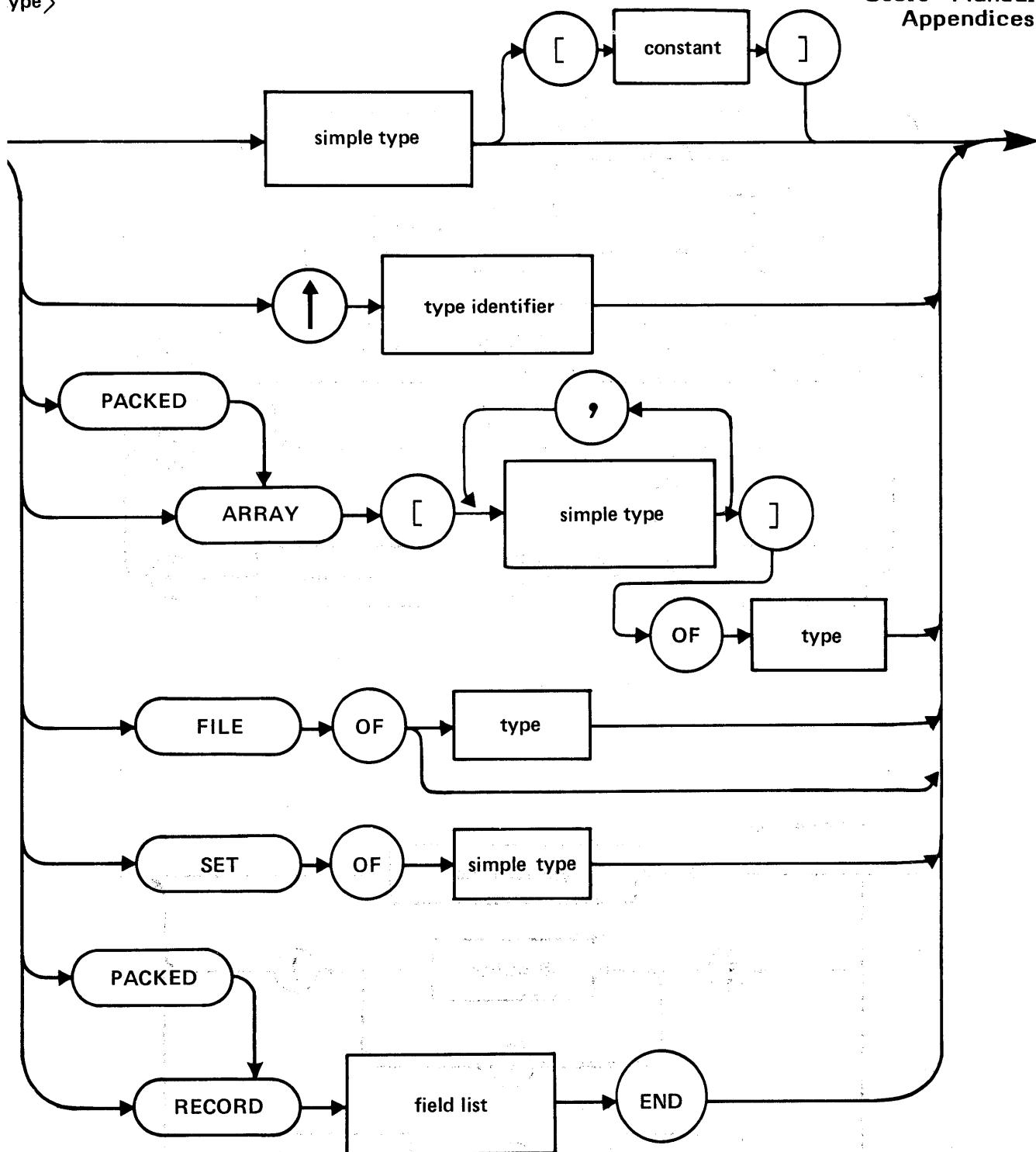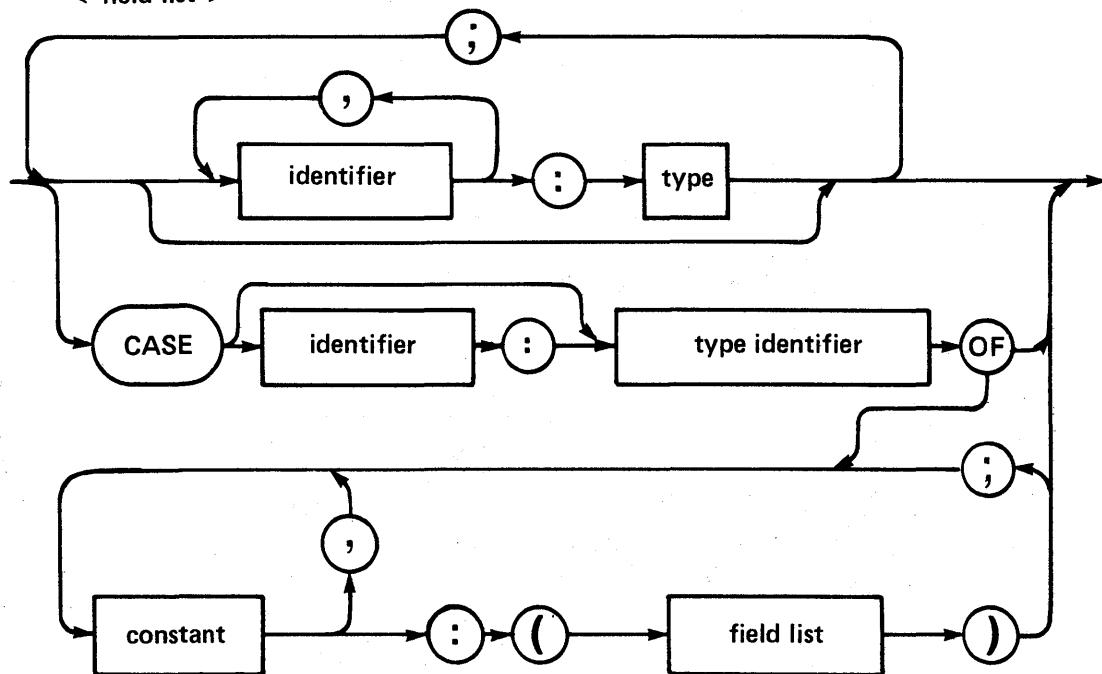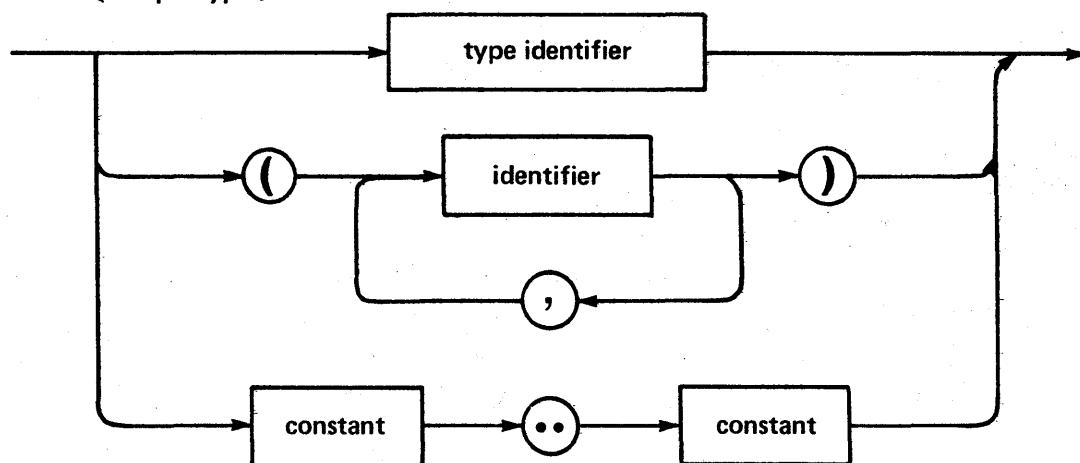
⟨constant definition⟩

⟨type definition⟩

⟨variable declaration⟩

< field list >

< simple type >

⟨routine⟩

⟨statement⟩

⟨expression⟩

⟨simple expression⟩

⟨parameter list⟩

⟨ factor ⟩
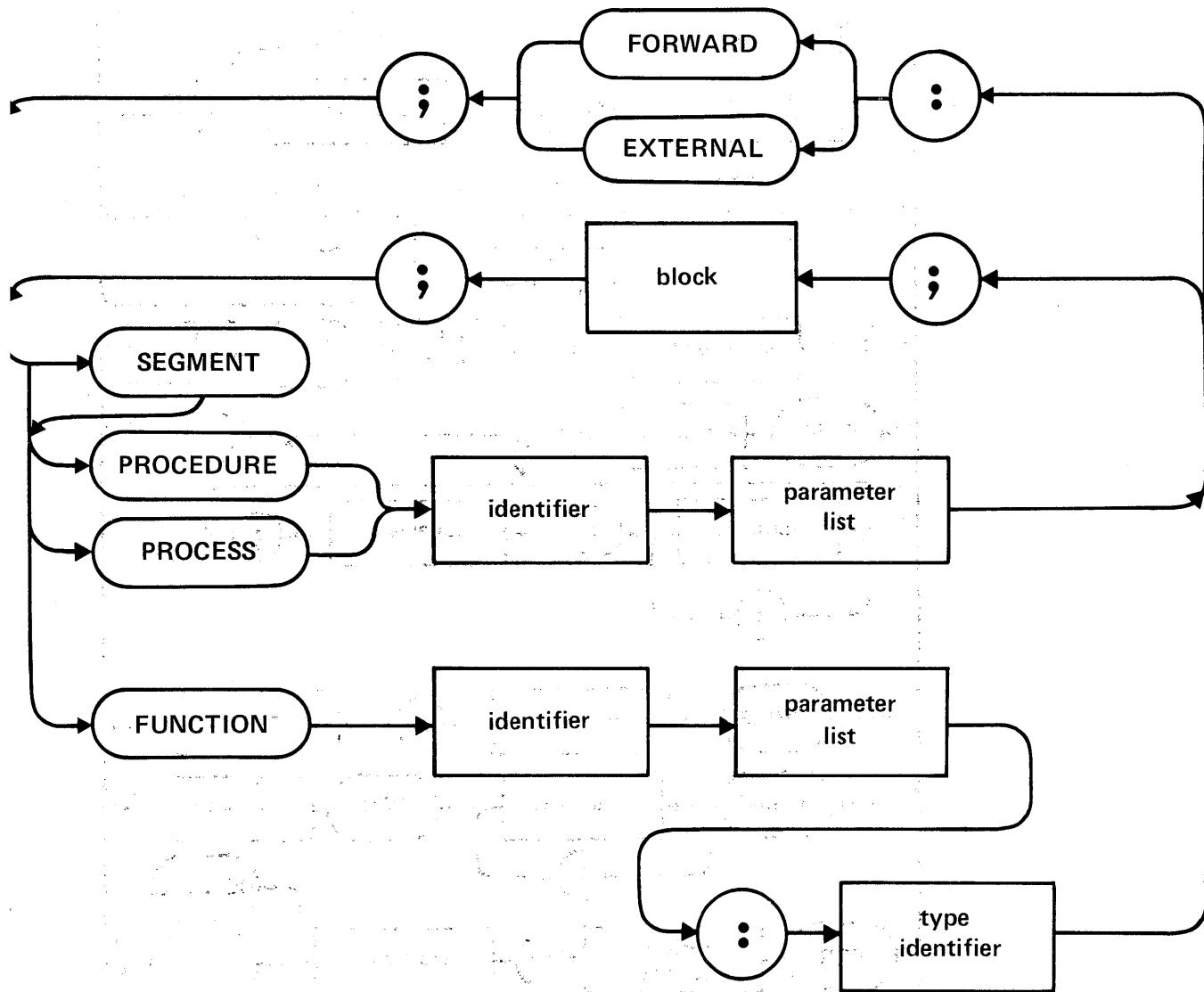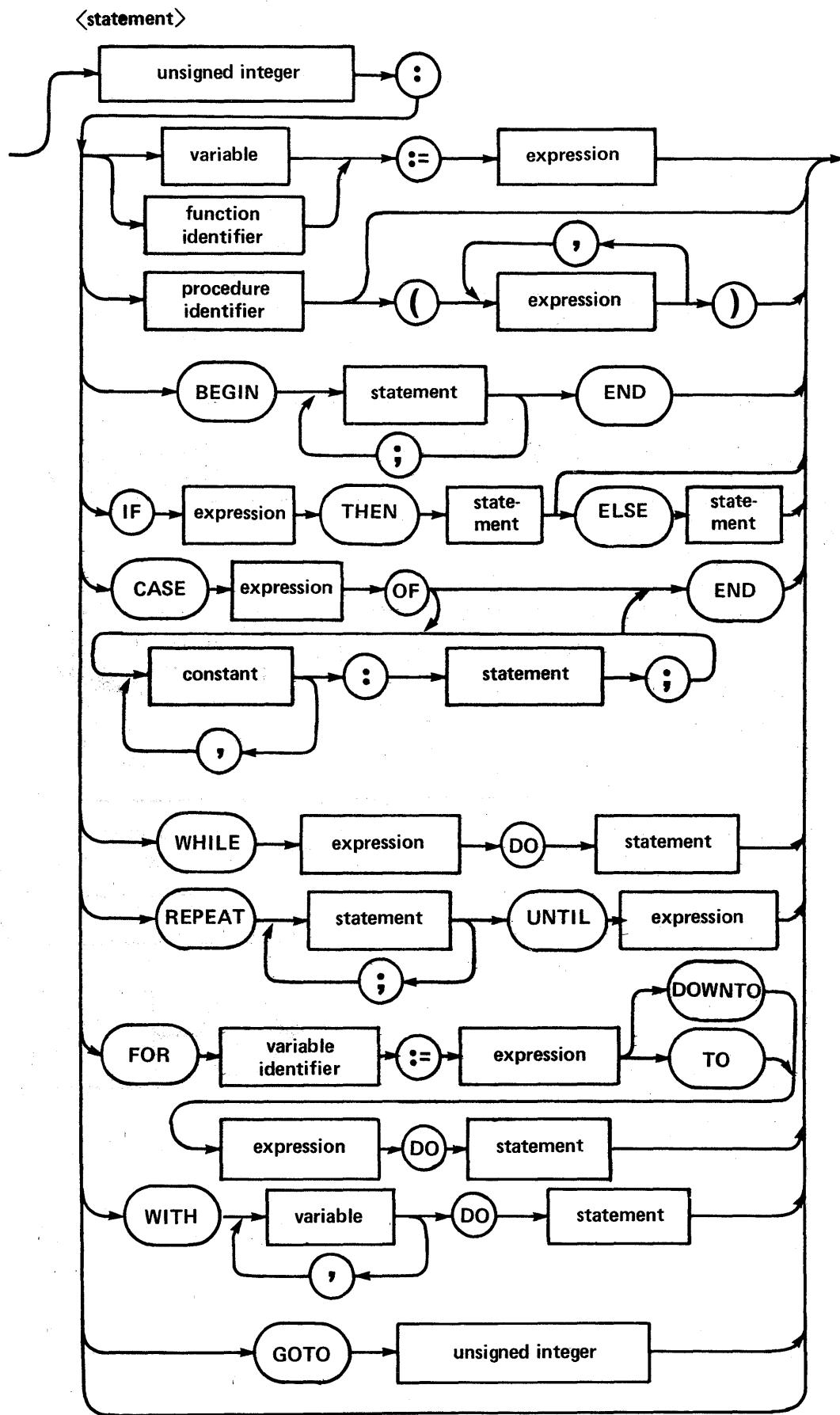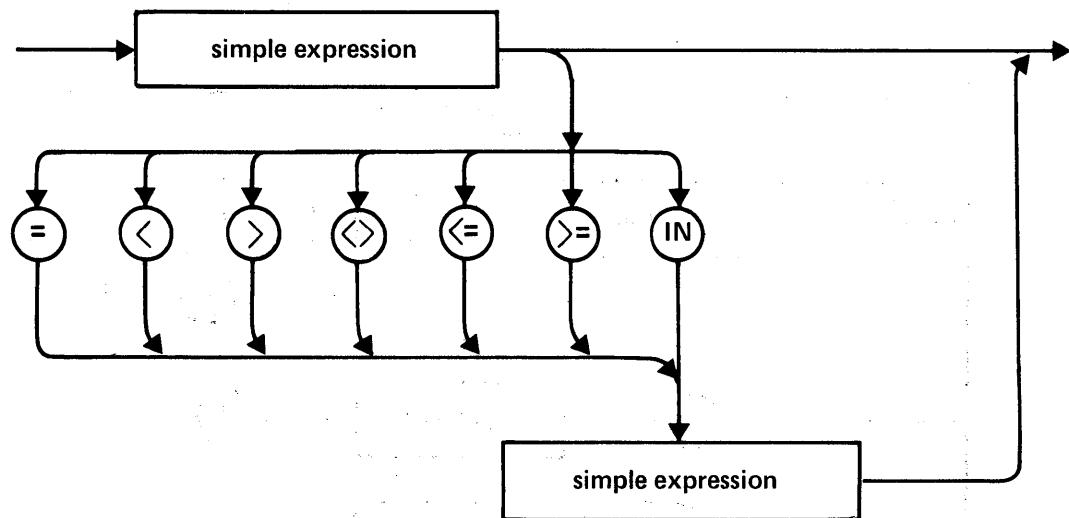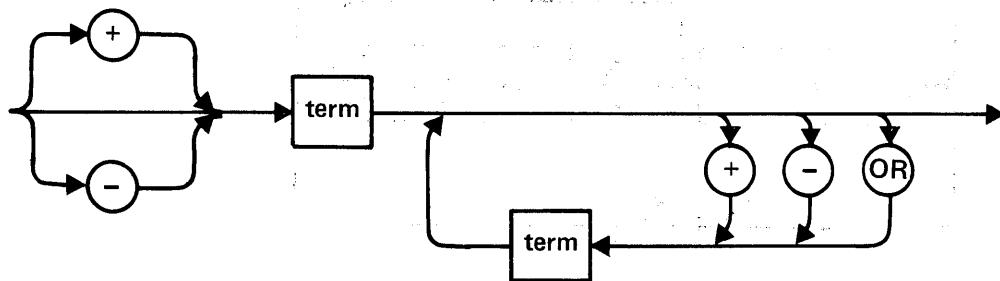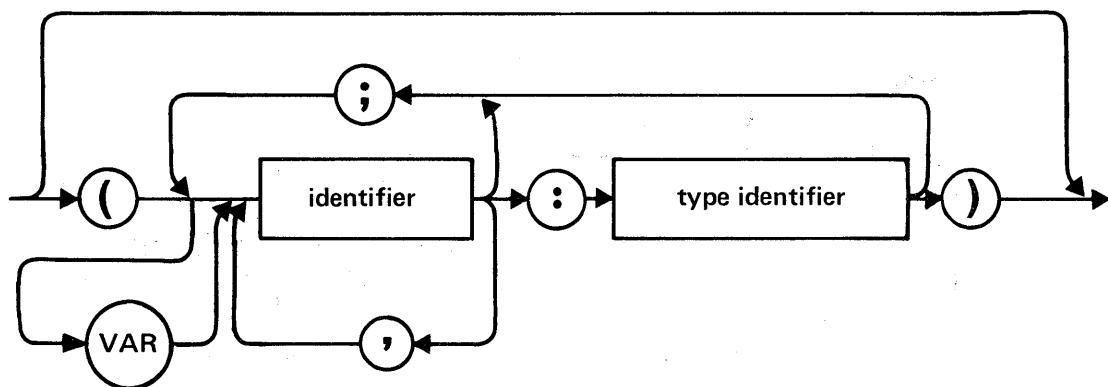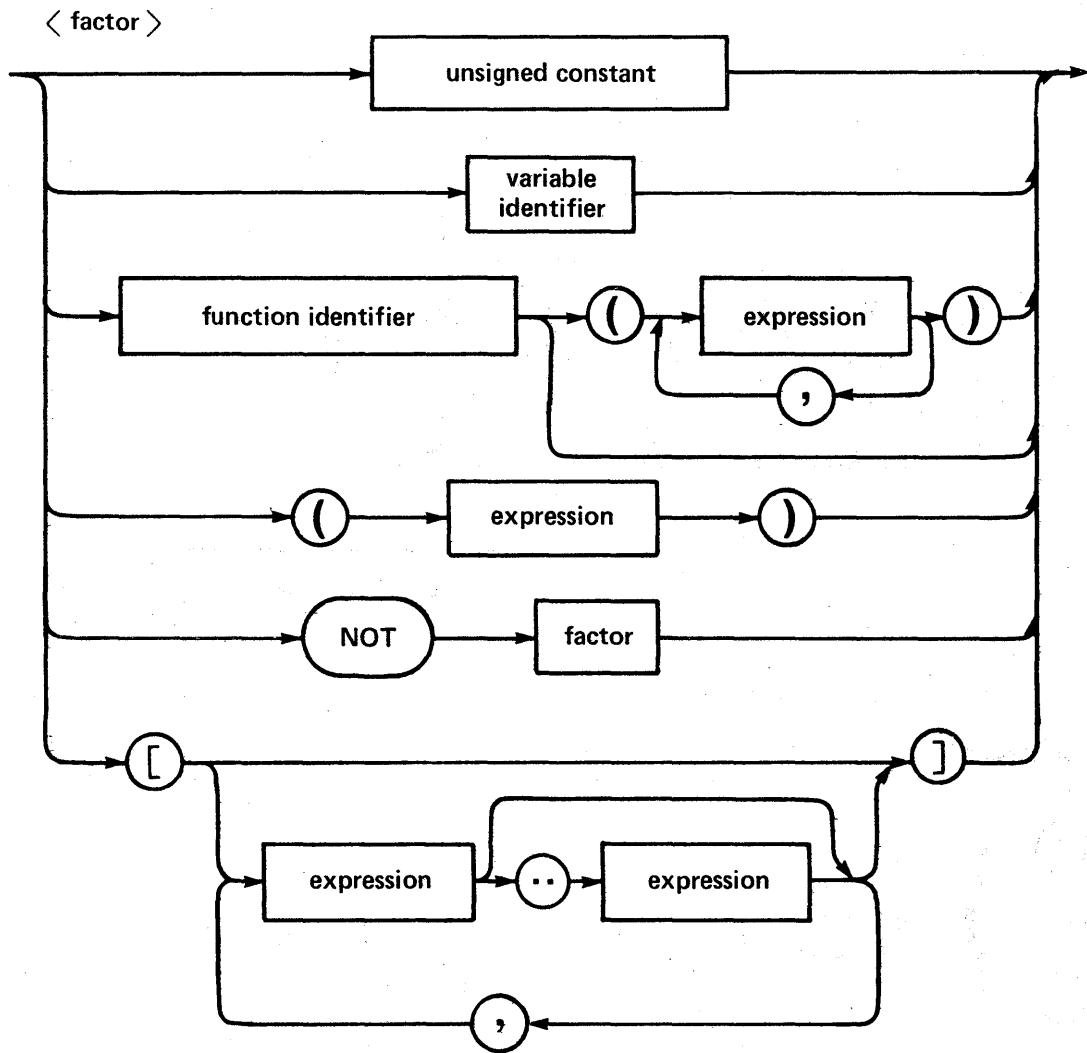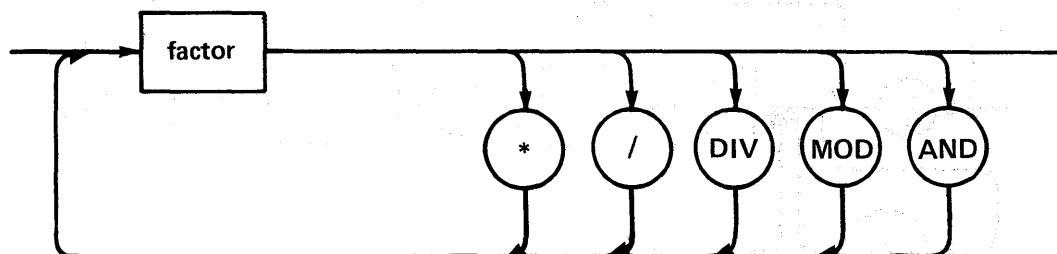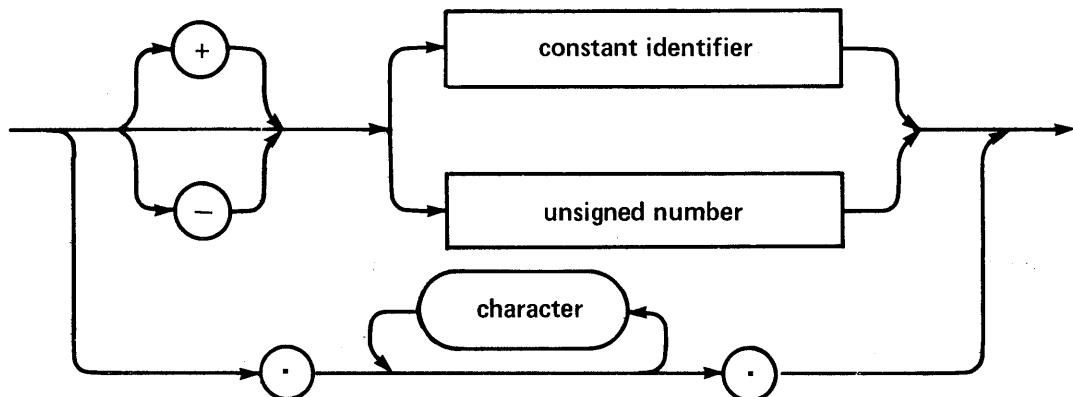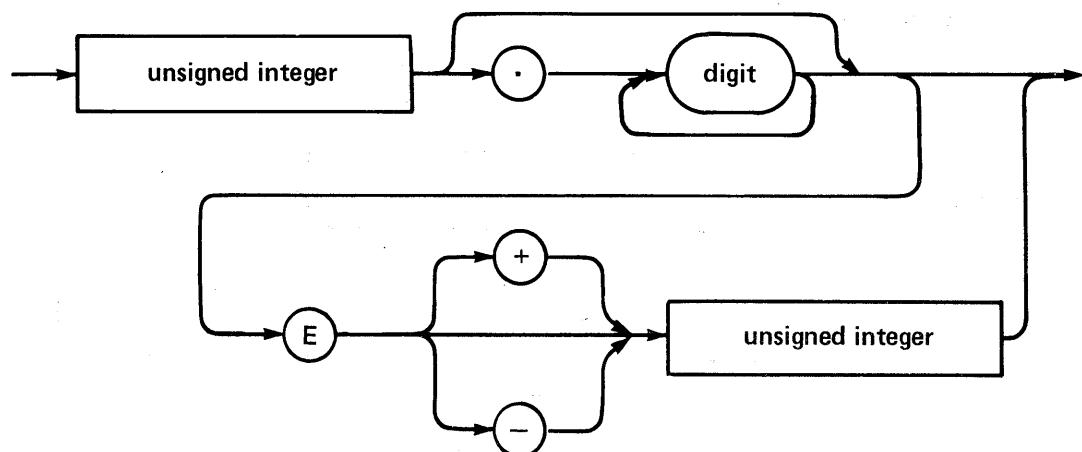
⟨ term ⟩

⟨constant⟩

⟨unsigned number⟩

⟨unsigned integer⟩
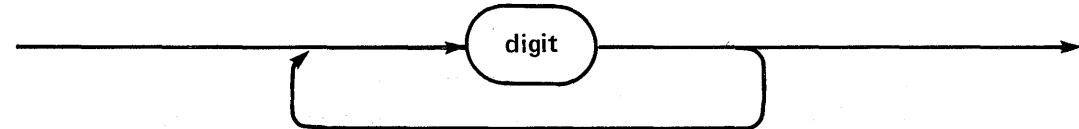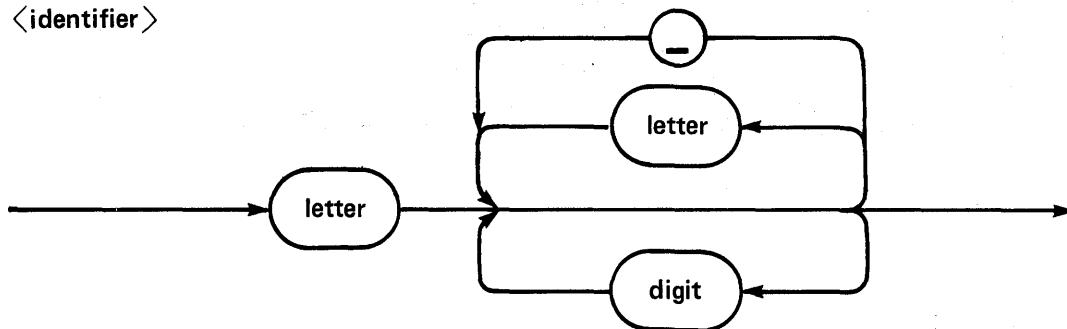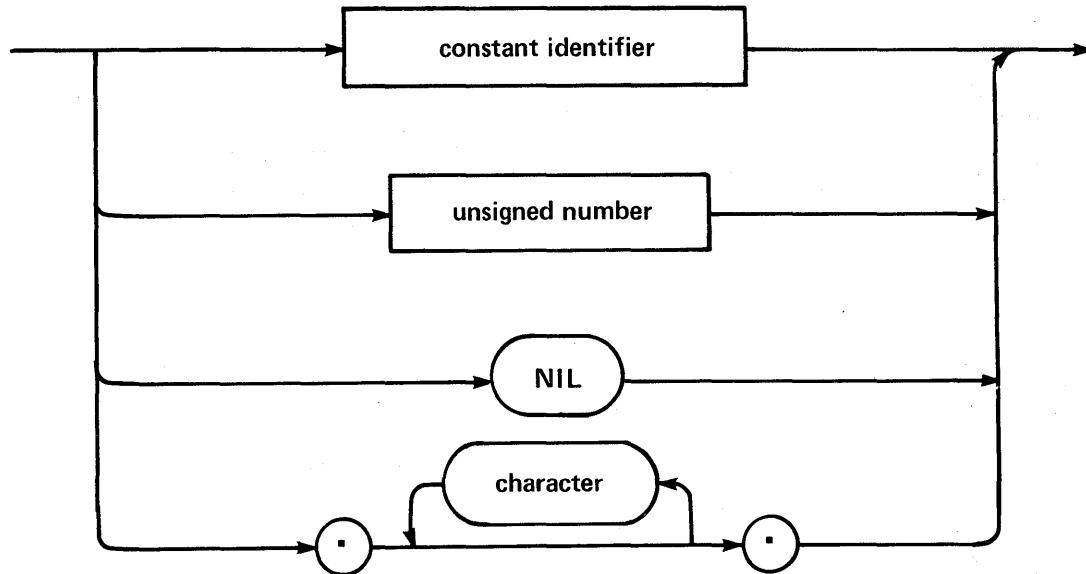
⟨identifier⟩

NOTE: The underscore character '−' is accepted but not significant

⟨unsigned constant⟩

XI.J   Appendix J -- Pascal Syntax Errors

1: Error in simple type
2: Identifier expected
3: unimplemented error
4: ')' expected
5: ': ' expected
6: Illegal symbol (terminator expected)
7: Error in parameter list
8: 'OF' expected
9: '(' expected
10: Error in type
11: '' expected
13: 'END' expected
14: ';' expected
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: error in <field-list>
20: '.' expected
21: '*' expected
22: 'INTERFACE' expected
23: 'IMPLEMENTATION' expected
24: 'UNIT' expected

50: Error in constant
51: ': =' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in <factor> (bad expression)
59: Error in variable

60: Must be of type 'SEMAPHORE'
61: Must be of type 'PROCESSID'
62: Process not allowed at this nesting level
63: Only main task may start processes

101: Identifier declared twice

102: Low bound exceeds high bound

103: Identifier is not of the appropriate class
104: Undeclared identifier
105: sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisified forward reference
118: Forward reference type identifier in variable declaration
119: Re-specified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: A forward declared function's result type can't be re-specified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be Boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar

149: Index type must be integer

150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substition of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was re-declared
172: Undeclared external file
173: FORTRAN procedure or function expected
174: Pascal function or procedure expected
175: Semaphore value parameter not allowed

182: Nested UNITs not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in INTERFACE section
185: Segment declaration not allowed in INTERFACE section
186: Labels not allowed in INTERFACE section
187: Attempt to open library unsuccessful
188: UNIT not declared in previous uses declaration
189: 'USES' not allowed at this nesting level
190: UNIT not in library
191: Forward declaration was not segment
192: Forward declaration was segment
193: Not enough room for this operation
194: Flag must be declared at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line

203: Integer constant exceeds range
204: 8 or 9 in octal number
250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
398: Implementation restriction
399: Implementation restriction

400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: 'PROGRAM' or 'UNIT' expected
406: Include file not legal
407: Include file nesting limit exceeded
408: INTERFACE section not contained in one file
409: Unit name reserved for system
410: disk error

500: Assembler error

XI.K   Appendix K -- American Standard Code for Information Interchange

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	´	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C		
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

INDEX

Note: In the following index, IG refers to the <u>Installation Guide,</u> and AG to the <u>Internal Architecture Guide.</u> Users interested in certain topics should refer to those manuals as well; this index does not attempt to reference them in detail (sorry). Users interested in FORTRAN or BASIC should refer to the appropriate manual.

Boldface indicates the principal description of an item.

Compiler	8, 16, 23, 32, 34, 41, 42, 157, 170, 203, **234-245**, 345
compile-time options	16, 17, 170, **236-245**
COMPRESSOR	19, 310, **364-367**
CONCAT	148, **195**
concurrent processes	144, 168, **353-361**, 417
conditional assembly	**288-290**
conditional compilation	237, 238, **244-245**
CONSOLE:	13, 29, 47, 49, 63, 64, 75, 87, 148, 150, 152, 234, 235, 241, 315, 376
control characters	92, 98-99, 111, 150, 1G
converting old source	**425-435**
C(opy	98, **103-104**, 106, 126
COPY	**196**
COPYDUPDIR	86, **383**
CP/M	1G
cross-referencer	see XREF
cursor	**89**, 92, 98-99, 102-126, 133, 1G
D(ate	**60**, AG
D(ebug	see DEBUGGER
DEBUGGER	201, **391-397**
DECODER	**376-382**
default disk	7, 12, 27, 28, 30, 49, 69, 79
D(elete	95, 98, 103, **105-106**, 126
DELETE	148, **197**
device numbers	12, 13, 26, 79, 403
devices	12-15, 26
directives	**264-287, 301-302**
directory	12, 36, 52, 53, 57, 58, 60, 61, 62, 64-66, 67, 71-72, 82-83, 84-87, 383-384, **AG**
disassembly	see DECODER
disks	see floppy disks
disk size	1G
disk space	1G
DISPOSE	**163**
DLE	45, 1G
E(dit	**3, 35**
Editor	1, 3, 8, 9, 10, 23, 24, 33, 35, 40, 42, **89-126**, 316, also see YALOE
EOF	29, **150**, 155, 161, 191, 217
EOLN	148, **151**, 155, 217

NOTES

SOFTech
microsystems

PROBLEM REPORT Date_____

Reported by_____

Address _____

Phone __(___)___-_____ x_____ Registration # _____

Notice: Your registration number is required in order for us to get a reply to you concerning the reported error. Please be as explicit as possible in describing the reported error, and if applicable, please attach a listing. Thank you.

☐ Please check here if a reply is not necessary.

Is this report a ____ software problem? ____ document problem? ____ design suggestion?

Document page #_____

Program affected (circle as many as apply): Version _____ _____ _____ _____ (e.g. II.0.A.3)

Editor, Filer, Operating System, Compiler (which language) _____,

Linker, Assembler (which processor) _____, Other (specify) _____

Did the system report an execution error? If so, what was the error #?_____

What was the text of the error message?_____

. . . and the P-code coordinates: S#_____ P#_____ I#_____?

What is the processor for your system? (Z-80, LSI-11, etc.)_____

Who is the manufacturer of your hardware? _____

What terminal do you use? _____

What is the problem?

What steps preceded the appearance of the problem?

What can we do to duplicate it?

Have you found a way to avoid the problem, and if so, what is it?

How seriously does this problem affect your projects?

Please feel free to add more pages if you run out of room.

For SofTech Microsystems' Use

Report number: _____ Impact: _____ Versions affected: _____

Symptom:

Error Description:

Problem:

Temporary Solution:

Fix:

Routing (for SofTech Microsystems' use)

1 ☐ to_____ for_____ Initial_____

2 ☐ to_____ for_____ Initial_____

3 ☐ to_____ for_____ Initial_____

—ALL-PURPOSE COUPON—

Pascal User's Group, c/o Rick Shaw
P.O. Box 888524
Atlanta, Georgia 30338 USA

NOTE

- Membership fee and All Purpose Coupon is sent to your Regional Representative.

- SEE THE POLICY SECTION ON THE REVERSE SIDE FOR PRICES AND ALTERNATE ADDRESS if you are located in the European or Australasian Regions.

- Membership and Renewal are the same price.

- Note the discounts below, for multi-year subscription and renewal.

- The U.S. Postal Service does *not* forward *Pascal News*.

		USA	Europe	Aust.
☐ Enter me as a new member for:	☐ 1 year	$10.	£ 6.	A$ 8.
☐ Renew my subscription for:	☐ 2 years	$18.	£10.	A$15.
	☐ 3 years	$25.	£14.	A$20.

☐ Send Back Issue(s)

☐ My new address/phone is listed below

☐ Enclosed please find a contribution, idea, article or opinion which is submitted for publication in the *Pascal News.*

☐ Comments: _____

_____

_____

$
ENCLOSED PLEASE FIND: A$
£____.____

CHECK no._____ .

NAME_____

ADDRESS_____

_____

_____

PHONE_____

COMPUTER _____

DATE _____

JOINING PASCAL USER'S GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.

- Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will *not* bill you.

- Please do not send us purchase orders; we cannot endure the paperwork!

- When you join PUG any time within a year: January 1 to December 31, you will receive *all* issues of *Pascal News* for that year.

- We produce *Pascal News* as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through *Pascal News*. We desire to minimize paperwork, because we have other work to do.

- *American Region* (North and South America): Send $10.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.

- *European Region* (Europe, North Africa, Western and Central Asia): Join through PUG(UK). Send £5.00 per year to: Pascal Users Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton S09 5NH, *United Kingdom;* or pay by direct transfer into our Post Giro Account (28 513 4000); International telephone: 44-703-559122 x700.

- *Australasian Region* (Australia, East Asia—incl. Japan): PUG(AUS). Send $A10.00 per year to: Pascal Users Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, *Australia.* International telephone: 61-02-23 0561 x435

PUG(USA) produces *Pascal News* and keeps all mailing addresses on a common list. Regional representatives collect memberships from their regions as a service, and they reprint and distribute Pascal News using a proof copy and mailing labels sent from PUG(USA). Persons in the Australasian and European Regions must join through their regional representatives. People in other places can join through PUG(USA).

RENEWING?

- Please renew early (before November and please write us a line or two to tell us what you are doing with Pascal, and tell us what you think of PUG and *Pascal News*. Renewing for more than one year saves us time.

ORDERING BACK ISSUES OR EXTRA ISSUES?

- Our unusual policy of automatically sending all issues of *Pascal News* to anyone who joins within a year means that we eliminate many requests for back issues ahead of time, and we don't have to reprint important information in every issue—especially about Pascal implementation!

- Issues 1-8 (January, 1974-May 1977) are *out of print.* (A few copies of issue 8 remain at PUG(UK) available for £2 each.)

- Issues 9-12 (September, 1977-June, 1978) are available from PUG(USA) all for $15.00 and from PUG(AUS) all for $A15.00

- Issues 13-16 are available from PUG(UK) all for £10; from PUG(AUS) all for $A15.00; and from PUG(USA) all from $15.00.

- Extra single copies of new issues (current academic year) are: $5.00 each—PUG(USA); £3 each—PUG(UK); and $A5.00 each—PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm. wide) form.

- All letters will be printed unless they contain a request to the contrary.

NOTES

NOTES