Personal Computer Computer Language Series

USERS' GUIDE

for the UCSD p-System™ Version IV.0

Produced by SofTech Microsystems, Inc.
Edited by Keith Shillington, Gillian Ackland, Randy Clark and Stan Stringfellow

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1982
© Copyright Regents of the University of California 1978
© Copyright SofTech Microsystems, Inc. 1979, 1980, 1981

UCSD, UCSD Pascal, and UCSD p-System are all trademarks of the Regents of the University of California.

CONTENTS

BOOK 1 OPERATIONS

CHAPTER 1. HARDWARE REQUIREMENTS 1-1
Configuration 1-3

CHAPTER 2. GETTING STARTED 2-1
Using the System 2-3
Powering-up with the p-System 2-4
Backing up your diskettes 2-5
Formatting a new diskette 2-5
Making backups 2-7
Important keys on the IBM Personal Computer Keyboard 2-9
Running the STARTUP: diskette 2-11

CHAPTER 3. MAKING USE OF THE p-SYSTEM 3-1
Making Use of the p-System 3-3
Software components 3-3
Operating System 3-4
Filer 3-4
Editor 3-5
Compilers 3-5
Libraries 3-8
Utilities 3-8
How to create and run a simple program 3-9
Error Messages 3-15
Ways to configure your diskettes 3-16
Large programs 3-21

CHAPTER 4. MAKING USE OF THE IBM PERSONAL COMPUTER HARDWARE ... 4-1
The Display ... 4-3
The Printer ... 4-4
Remote Devices ... 4-5
Making Use of IBM Personal Computer Hardware ... 4-6
Dual-Sided Diskette Drives ... 4-7
Formatting Your Diskettes ... 4-7
Creating System Diskettes ... 4-7
Creating Storage Diskettes ... 4-9
Serial Printer Support ... 4-10
Connecting Your Printer ... 4-11
Reconfiguring Your System Diskette ... 4-12
RS232SET Utility ... 4-15

APPENDIX A. THE p-SYSTEM PACKAGE... A-1
Documents shipped with the p-System ... A-3
Diskettes and files shipped with the p-System ... A-4

CHAPTER 1. HARDWARE REQUIREMENTS

Contents

Configuration 1-3

NOTES

CONFIGURATION

To run the p-System on the IBM Personal Computer, your IBM Personal Computer configuration must include:

- At least 64K bytes of main memory.
- Two diskette drives.
- IBM Personal Computer Monochrome Display *or* video monitor or color TV with RF modulator.

NOTES

CHAPTER 2. GETTING STARTED

Contents

Using the System 2-3
Powering up with the p-System 2-4
Backing up your diskettes 2-5
Formatting a new diskette 2-5
Making backups 2-7
Important keys on the IBM Personal
Computer Keyboard 2-9
Running the STARTUP: diskette 2-11

NOTES

Using the System

Now that you have an IBM Personal Computer and a UCSD p-System, you are probably wondering how to use them. Once you have become familiar with the IBM Personal Computer hardware, and are ready to use the p-System, you should read this Operations Guide FIRST.

We are going to talk about three things:

1) How to power up the p-System and begin using it.

2) Ways of using the p-System effectively.

3) How the p-System uses some of the IBM Personal Computer hardware.

It is important that you read through this Operations Guide. If you don't understand some of it, you may be in danger of LOSING programs. If you have programmed before, you know that means losing a lot of time (and maybe even money, if you happen to destroy the diskettes on which your p-System is shipped).

Moral: keep reading.

(We do occasionally mention a topic for more experienced p-System users. But essentially you should read this Operations Guide cover-to-cover.)

Enough preaching. You're eager to get your p-System running. That's what the next section tells how to do.

Powering-up with the p-System

First, take a look at the Appendix in the back of this Operations Guide. It lists all of the documents and diskettes that you should have received with the p-System. Make sure that all of it is there for the p-System you have purchased (Pascal, or FORTRAN, or both).

There are three diskettes that will "bootstrap" on the IBM Personal Computer. For now, we need only one. "Bootstrapping" means bringing the p-System up: most of this work is done by the System itself, hence the term (as in "pulling yourself up by the bootstraps").

Take the diskette that is labeled SYSTEM2:. If your IBM Personal Computer is on right now, turn it off, and wait at least 6 seconds. Put SYSTEM2: in the LEFT-HAND diskette drive. The label should be facing up, and the oblong slot that exposes the diskette itself should be pointed away from you. Close the small door. Now turn the power on.

Don't panic if it takes a while! You should hear the speaker beep, then the light should come on in front of the left-hand diskette drive, and you should hear the diskette drive reading information from the diskette. After about a minute, if everything goes well, the IBM logo will appear on the display, and above it will be the main promptline of the p-System.

The promptline is the outer level of the "Operating System," and it shows most of the major p-System commands. For now, you don't need to worry about what they mean (you will get to see how some of them are used in the next section).

If you don't get a promptline after two minutes, then turn off the IBM Personal Computer. Read over the preceding few paragraphs and make sure you did

everything correctly (be sure you are using the SYSTEM2: diskette!). Then try it again. ALWAYS wait at least 6 seconds between turning off the IBM Personal Computer and turning it on again.

If you continue to have trouble, stop right here and call your dealer. Resume at this point once you have bootstrapped your p-System (or, if you like, read ahead while you are waiting for help, and learn what to expect).

When you bootstrap for the first time with a color card, and the dipswitches on your computer are set to default to the 40-column mode, the screen you see is only the right-hand half of the full p-System display. Later on in this chapter, we will tell you how to deal with this.

If the dipswitches on your computer are set to default to the 80-column mode, you will see the full screen when you bootstrap.

All right, now you have bootstrapped your p-System. The next thing to do is play it safe and make backup copies of ALL your diskettes.

Backing up your diskettes

Formatting a new diskette

A brand new diskette must be formatted before you can use it on the IBM Personal Computer. This is done by running a small "utility program" called DISKFORMAT.CODE.

A "utility program" is a program that comes with the p-System, and is run like any user's program.

The p-System is shipped on six or seven diskettes depending on what you have purchased, and you must make a backup copy of each of them. The first step is to format seven new diskettes.

For your protection, the p-System diskettes that are shipped cannot be written on. Making backups is necessary. It allows you to use your p-System more freely, and to configure the files on your diskettes in a more convenient manner, as we describe in the next chapter.

When using the p-System, the left-hand drive on the Personal Computer is referred to as "drive #4", and is always used to bootstrap the p-System. The right-hand drive is referred to as "drive #5".

We assume that you have bootstrapped your p-System, and are looking at the main promptline. To run DISKFORMAT.CODE, place the diskette called UTILITY: in the right-hand diskette drive, and type "X" for eX(ecute. The p-System will ask you:

Execute what file?

Now type "#5:DISKFORMAT" and then press the *enter* key. (The *enter* key is to the right of the alphabet keys, and looks like an arrow pointing down and to the left.)

DISKFORMAT should reply with the following prompt:

Enter unit number of diskette to be formatted (4..5):

REMOVE both the diskette SYSTEM2: and the diskette UTILITY:. Put a BLANK diskette into the right-hand drive, and type "5" followed by *enter.*

DISKFORMAT will respond:

Insert disk in unit 5 and press <enter> ...

Since you have already put the blank diskette in the right-hand drive (drive #5), simply press *enter*. DISKFORMAT takes a while to run, and informs you when it has finished.

Once you have formatted a diskette, you have five or six more to go. Once DISKFORMAT is finished, type "U" for U(ser-restart to start DISKFORMAT over again. Put another blank diskette into drive #5, and answer DISKFORMAT's prompts in the same way we have shown.

If DISKFORMAT gives you an error message while trying to format a diskette, there is probably something wrong with the diskette. Set it aside and try another.

Now that you have formatted all your diskettes, it is time to back up the p-System diskettes.

Making backups

You should put SYSTEM2: (the disk you bootstrapped with) back into drive #4. Now type "F" for F(iler. This command puts you at another "level" of the p-System, where the letters you type are now commands to the Filer.

Type "T" for T(ransfer. The filer will ask you:

Transfer what file?

Now REMOVE the System diskette, place one of the six or seven p-System diskettes in the left-hand drive (drive #4) and one of the formatted blank diskettes in the right-hand drive (drive #5).

Answer the prompt with "#4,#5", and press the *enter* key.

The Filer then asks:

Transfer 320 blocks ? (Y/N)

Type "Y", and press the *enter* key. (320 blocks is the size of the entire diskette.)

The Filer will transfer the contents of the diskette in drive #4 onto the diskette in drive #5. When it is finished, it displays a message to that effect.

When the transfer is complete, remove both diskettes, label the new backup copy appropriately, type "T" again, and repeat these steps (placing your original diskette in drive #4, and your backup in drive #5) until you have made backup copies of all your p-System diskettes.

> **Warning:** If you put the diskettes in backwards (i.e., the blank diskette in drive #4 and the good diskette in drive #5), the Filer will display a message such as:
>
> **Destroy SYSTEM2: ?**

You should type "N" or "n", and correct the mistake.

You are now ready to use your backup copies for everyday work. Keep the diskettes that were originally shipped in a safe place.

Important keys on the IBM Personal Computer Keyboard

When you run your p-System, there are some important keys that you should know about. This section describes them briefly. You probably will not want to absorb all of the information here on first reading, but do read through the section, and refer back to it when you need this information.

First, there is the *enter* key (it looks like an arrow pointing down and to the left). Whenever you answer a prompt in the p-System (except for single-letter commands), you should follow it with an *enter*. When you are using the Editor, the *enter* key acts as a carriage return on a typewriter.

To type all capital letters, you can press the CAPS LOCK key (as on a typewriter keyboard). To type lower case letters again, press CAPS LOCK a second time.

Once the p-System has been bootstrapped, it is not necessary to turn off the IBM Personal Computer in order to bootstrap it again. You can insert the diskette you want to bootstrap from in drive #4 (or leave it there if it is already there), hold down the Ctrl and ALT keys, and press the "DEL" on the numeric pad. The IBM Personal Computer will re-bootstrap your p-System. Since you need to press three keys at once, using two hands, the danger of bootstrapping accidentally is negligible.

When a program is sending output, it is possible to stop and start that output by holding down the Ctrl key and pressing "S". The first time you type Ctrl S, output stops, the next time you type it, output begins again, and so forth.

If you no longer want the program to send any output, hold down the Ctrl key and press "F". F stands for "flush". The program will continue to run, but will not send any output, and should soon terminate.

If something goes wrong while a program is running, and you wish to halt it completely, holding down the Ctrl key and pressing "@" causes a BREAK. The p-System displays an error message, then re-initializes itself. Another way to halt a program is to hold down Ctrl and press BREAK (the BREAK key is the same as the SCROLL LOCK key). This causes a hardware BREAK, which is more difficult for the p-System to recover from. The rule of thumb is: use Ctrl @, and if that fails, use Ctrl BREAK. After a Ctrl BREAK, it will often be necessary to re-bootstrap.

When you are typing input to the p-System, you can correct errors by backspacing over them. The *backspace* key is on the top row, and looks like an arrow pointing left. You can also erase an entire line of input by typing Ctrl *backspace*.

The vector keys (four arrows) occupy the same keys as the numeric pad. Normally you will be using the vector keys. To use the numeric pad as a numeric pad, press NUM LOCK. This works in much the same way as CAPS LOCK. To use the vector keys again, press NUM LOCK a second time.

The vector keys are usually used within the p-System Editor. They are used to move the cursor, along with *backspace, enter, tab,* and some other commands (the cursor is the flashing underline that appears on most screens).

When the Editor is in eX(change mode, INS inserts a single space, and DEL deletes a single character.

Another character you must use while in the Editor is called *etx*. This is the character that accepts insertions and deletions. To send an *etx*, hold down the Ctrl key and press "C". If you do NOT want to accept an insertion or deletion, you may press the ESC key (for "escape").

If the display you are using is in half-screen mode (40 columns of characters), then you can use the *left-arrow* and *right-arrow* keys to view the hidden portion of the display. The rule is that the p-System always displays the columns that surround the cursor.

That should be enough information for now.

Running the STARTUP: diskette

If you have never used the p-System before then we recommend that you become familiar with it by using Ken Bowles' Beginners' Guide for the UCSD p-System (which is supplied with all p-Systems). This guide uses a number of programs, and these programs are supplied on the diskette labeled STARTUP:.

STARTUP: can be bootstrapped. It contains a p-System with an Editor, but not a Compiler. To use a Compiler, see Chapter 3 in this Operations Guide. STARTUP: also contains the following files:

NAMEFILE

SCDEMO.CODE

COPYSCUNIT.CODE

UPDATE.CODE

COMPDEMO.TEXT

EDITDEMO.TEXT

UPDATE.TEXT

The use of these files is explained in the Beginners' Guide.

When you bootstrap on the STARTUP: diskette, you will find yourself in the middle of a program. This is a demonstration program that is meant to give you some of the feel of using the p-System. The Beginners' Guide tells you how to use it and what to expect.

CHAPTER 3. MAKING USE OF THE p-SYSTEM

Contents

Making Use of the p-System 3-3
Software components 3-3
Operating System 3-4
Filer 3-4
Editor 3-5
Compilers 3-5
Libraries 3-8
Utilities 3-8
How to create and run a simple program 3-9
Error Messages 3-16
Ways to configure your diskettes 3-15
Large programs 3-21

NOTES

MAKING USE OF THE p-SYSTEM

This chapter should be more interesting than the previous chapter, since it describes the major components of the p-System, and some of the more effective ways to use them.

Software components

What is a p-System, anyway? It's a collection of software components that come in the form of files saved on diskettes. This software is used for writing and running programs. The programs you write will use the IBM Personal Computer's hardware in various ways.

When you use the p-System, you begin by bootstrapping it. When you have bootstrapped, you see a System promptline that looks like this:

```
Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem,? [IV.02 B3j-A]
```

Each capital letter shown on the promptline is a command to the p-System. Typing one of these letters causes something to happen. Some of the commands, such as E(dit and F(iler, call other programs that in turn have their own promptlines and their own set of commands.

In other words, each component of the p-System serves a particular purpose. All of them are fairly self-explanatory, and all of them are meant to be used by a single user sitting in front of the IBM Personal Computer display (in other words, "interactively").

This section gives some brief descriptions of the major components of the p-System.

Operating System

Very little of the Operating System is actually seen by the user. The Operating System is essentially the program that controls the IBM Personal Computer's resources and calls various other programs.

The part of the Operating System that IS important to the user is the main promptline: the one you see when the p-System is first bootstrapped. This set of commands allows you to call other portions of the p-System, and to run programs that you have written yourself.

The Operating System is called SYSTEM.PASCAL. It cannot run unless the files SYSTEM.MISCINFO, SYSTEM.INTERP, and the invisible bootstrap code are also on the disk that you bootstrap. These files all appear on the diskettes STARTUP:, SYSTEM2:, and SYSTEM4:, which are shipped with the p-System. Any ONE of these diskettes can bootstrap the p-System, when placed in the left-hand disk drive, as described in Chapter 1.

The Operating System and its commands are described in Chapter 2 of the *User's Guide for the UCSD p-System.*

Filer

You have already used the Filer briefly to make backup copies of your p-System diskettes. The Filer is used for maintaining the collection of files on a diskette, and transferring them from one location to another.

SYSTEM.FILER is the Filer program. It is shipped on STARTUP:, SYSTEM2:, and SYSTEM4:.

The use of the Filer is described in Chapter 3 of the User's Guide.

Editor

The Editor allows you to create new files such as programs or documents. It also allows you to modify old files. The Editor makes use of the entire display screen, so it is easy to see the text you are working on, and modify it as necessary.

The Editor is called SYSTEM.EDITOR, and it is also shipped on the three bootstrap disks.

Using the Editor is described in Chapter 4 of the User's Guide.

Compilers

With the UCSD p-System on the IBM Personal Computer, you can program in UCSD Pascal or in FORTRAN. Programming in one of these languages means creating a text file with the Editor, and then "compiling" it by calling the appropriate compiler. The compiler translates the program text into a form that the p-System can execute on the IBM Personal Computer.

There are two UCSD Pascal compilers, and two p-Systems. The ONLY difference between the two is that one set uses real numbers that are two words long (32 bits), and one set uses real numbers that are four words long (64 bits). If you don't understand the difference, for now you only need to remember that four-word real numbers are more accurate.

The UCSD Pascal compilers are shipped on the diskette PASCAL:. The two-word compiler is called SYSTEM.COMPILER, and the four-word compiler is called PASCAL4.COMPILE. The p-System that supports two-word reals is the SYSTEM2: diskette that you have already bootstrapped. The p-System that supports four-word reals is on the

SYSTEM4: diskette. The ONLY difference between these two System disks is the size of their real numbers.

We recommend that if you use real numbers, you use four-word reals (although the two-word reals run somewhat faster). Since these are more accurate, we will eventually phase out two-word reals.

When you compile a program in UCSD Pascal, the name of the compiler file must be SYSTEM.COMPILER. The p-System is shipped with the two-word Pascal compiler named SYSTEM.COMPILER. If you intend to use Pascal with four-word reals (PASCAL4.COMPILE) or one of the FORTRAN compilers (FORTRAN2.CODE or FORTRAN4.CODE), you must use the Filer to change file names. Change the name of SYSTEM.COMPILER to PASCAL2.COMP, and the name of the compiler you wish to use to SYSTEM.COMPILER.

Also, when you compile a UCSD Pascal program, the file SYSTEM.SYNTAX should be on the System disk (the diskette you bootstrap with). This file contains all the error messages that the Pascal compiler needs if (alas) it encounters a syntax error while compiling your program. SYSTEM.SYNTAX is shipped on SYSTEM2: and SYSTEM4:.

The UCSD Pascal language is described in the *PASCAL Reference for the UCSD p-System.*

The FORTRAN compilers are on the FORTRAN: diskette. As with UCSD Pascal, there are two of them. FORTRAN2.CODE has two-word real numbers and is used along with FORTLIB2.CODE. FORTRAN4.CODE has four-word real numbers, and is used along with FORTLIB4.CODE.

When you compile a FORTRAN program, the name of the compiler you use must be changed to SYSTEM.COMPILER (as described above), and the name of the matching library must be changed to SYSTEM.LIBRARY in a similar fashion. We will say more about this in the section below on "Ways to configure your diskettes."

FORTRAN does not have a file that corresponds to Pascal's SYSTEM.SYNTAX.

Information on the FORTRAN language, and using the LIB files that accompany each FORTRAN compiler, may be found in the *FORTRAN-77 Reference for the UCSD p-System.*

It is also possible to program directly for the IBM Personal Computer's 8088 processor. These programs are said to be "assembled" rather than compiled, and the p-System program that does this is SYSTEM.ASSMBLER.

SYSTEM.ASSMBLER is shipped on the diskette EXTRAS:, along with the files 8086.OPCODES and 8086.ERRORS. These two files must be present whenever the assembler is run.

We expect that only programmers already familiar with assembly language will attempt to use SYSTEM.ASSMBLER, and then only when they have a good reason to. It is far easier to program in a "high-level" language such as UCSD Pascal.

When you write a UCSD Pascal or FORTRAN program that calls assembly-language routines (yes, it's possible), you will need to use the Linker program, which is shipped on the EXTRAS: diskette as SYSTEM.LINKER.

The assembly language for the 8086/87/88 processor is described in the *Assembler Reference for the UCSD p-System.*

Libraries

Frequently, a number of programs will need to perform the same operations. It is not necessary to "re-invent the wheel" and write the same code over for every program that needs it. The programmer can write a portion of code called a "UNIT," and compile it separately from the programs that use it. A disk file that contains one or more UNITs for use by several programs is called a "library".

The most important library is called SYSTEM.LIBRARY, and such a file appears on both SYSTEM2: and SYSTEM4:. SYSTEM.LIBRARY contains the units IBMSPECIAL and TURTLEGRAPHICS, which allow you to use certain features of the IBM Personal Computer hardware. It also contains the unit LONGOPS, which is the code that performs operations on UCSD Pascal long integers.

A description of IBMSPECIAL and TURTLEGRAPHICS, and how to create your own library, is described in the User's Guide, Chapter 5. That chapter also describes the utility LIBRARY, which is used for creating and maintaining software libraries.

Utilities

We have just mentioned the utility LIBRARY, and in Chapter 2 we used the utility DISKFORMAT. Utilities are programs, and they are run in just the same way as a user's program. They are shipped with the p-System, and in general, they provide important services, but are not used as frequently as the programs (like the Filer, Editor, or Compiler) that are called directly from the Operating System.

Most of the p-System utilities are shipped on the diskette UTILITY:. A few of them are shipped on EXTRAS:. They are described in the Users' Guide, Chapter 7.

EXTRAS: also contains some files to be used by programmers who are a bit more experienced. We will talk about these later.

How to create and run a simple program

In this section, we will "walk you through" creating a simple program, compiling it, and running it.

(FORTRAN programmers note: this sample uses a brief Pascal program. What we are really trying to illustrate is the use of the Editor to create a workfile, the use of the R(un command to compile it, and the use of the Filer to change its name. All these aspects of the p-System are the same whether you are programming in Pascal or in FORTRAN.)

First, bootstrap the p-System if you haven't already done so. (We assume that you are a new user who has not yet created a workfile. If you do have a workfile, you should save it before going through this demonstration.)

Now type "E" for E(dit. You should get a display that looks like this:

```
>Edit:
No workfile is present. File? ( <ent> for no file )
```

Since we are creating a new program, press *enter* to continue.

THE p-SYSTEM

Now type "I" for I(nsert, and type the following text as shown. To end a line, press the *enter* key (as you would use the carriage return on a typewriter). Notice that once you have indented a line, the lines that follow it are indented automatically:

```
PROGRAM EASY;
  BEGINN
    WRITELN('HELLO, THERE!');
    WRITELN;
    WRITELN('YOU'VE JUST RUN YOUR FIRST PROGRAM'.);
  END
```

If you make a mistake while typing, *backspace* over it and re-type the correct characters. When you have finished typing in the program, type Ctrl C (you must hold down the Ctrl key and then press "C"). This should get you out of I(nsert, and back to the Editor's main promptline.

Ah, but the program we made you type contains an error (if you didn't type it just the way it's shown here, it may contain more than one!). The "BEGIN" in the second line should contain one "N", not two.

To remove this extra letter, move the cursor (that's the flashing underline) back to the FIRST "N" by using the "cursor keys" or "arrow keys." These are the four arrows that appear on the numeric pad on the right-hand side of the keyboard. Try moving around your program until you get the feel for it. The *enter* key, *backspace* key, and *tab* key can also be used to move the cursor.

If you try to use the cursor keys, and get a message that looks something like this:

```
ERROR: Repeatfactor > 10,000
    Please press <spacebar> to continue.
```

... don't panic! This simply means that the NUM LOCK key has been pressed, and you were typing numbers instead of typing cursor arrows. To use the

cursor keys, press *space* , then press NUM LOCK, and the cursor keys should work again.

Now, is the cursor at the first "N" in the second line? Good. To remove the "N", press "D" for D(elete, then type a space. This should delete the first "N". Then type Ctrl C, which should close up the line and return you to the main Editor prompt. If this worked, we can go on.

Type "Q" for Q(uit (sound familiar?), and you will see a prompt that looks like this:

```
>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
```


Type "U" for U(pdate. What this does is create a temporary file called SYSTEM.WRK.TEXT. This file is on your System disk, and it contains the text that you just typed in. It is referred to as the "workfile," and a little later we will show you how to save it under a different name.

Now that we have a program, we want to run it. But we have to compile it first. To do so, take YOUR backup copy of the disk PASCAL:, and put it in the right-hand disk drive. Type "R" for R(un. The Operating System will recognize that your workfile hasn't been compiled, and so it will call the compiler SYSTEM.COMPILER that is on the disk in drive #5. When the compiler is finished, the Operating System will run your program. While the compiler is compiling, it displays some progress information:

```
Pascal compiler - release level IV.0 c2-4
<    0>..
EASY
<    2>...
   5 lines compiled

EASY
```

... and when it is finished, it should run your program, creating some output that looks like this:

```
Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem,? [IV.02 B3j-A]
Running...
HELLO, THERE!

YOU HAVE JUST RUN YOUR FIRST PROGRAM.
```

The compiler creates another portion of the workfile called SYSTEM.WRK.CODE. As long as SYSTEM.WRK.CODE exists, using the R(un command will run your program over again without re-compiling it.

(There is little difference between the expressions "running a program" and "executing a program." But there is a large difference between the Operating System command R(un and the Operating System command eX(ecute, as you will discover if you read Chapter 2 of the Users' Guide.)

If there had been a "bug" or mistake in the program you were compiling (and there might be, if you typed it differently from the program in this booklet), the compiler would display a message something like this:

```
Compiling...

Pascal compiler - release level IV.0 c2-4
<    0>..
EASY
<    2>..
      WRITELN;
      WRITELN('YOU'VE <---
Illegal symbol (terminator expected)
Line 5
Type <sp> to continue, <esc> to terminate, or 'e' to edit
```

When the compiler tells you there is an error, there are three things you can type:

1) ESC ends the compilation;

2) *space* or *enter* continues the compilation and allows you to see what other bugs the Compiler might report;

3) "E" or "e" gets you back into the Editor and allows you to fix the bug on the spot.

Now, we could show you more about the Editor, but the introduction to Chapter 4 in the Users' Guide does a good job of that. So for now, we will just save the workfile that you have created.

Go back into the Filer (by pressing "F" at the main promptline). Now type "S" for S(ave. The filer will ask you:

Save as what file ?

Type the filename "TESTING", and then press *enter*. When the workfile has been saved, we get this message:

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
Text file saved & Code file saved

The files SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE are renamed TESTING.TEXT and TESTING.CODE, respectively. These files are now your current workfile.

Notice that we didn't need to type ".TEXT" or ".CODE". The Filer supplies these suffixes. This is a convenient shortcut, but in the Filer it only applies to the workfile commands G(et and S(ave. For any other Filer command, you must type the entire filename.

Now that the TESTING files are considered the workfile, we can go back into the Editor (you should also be able to do this by now), and the Editor will automatically read in TESTING.TEXT for you to work on.

And so on.

When you get tired of your TESTING files, go into the Filer and type "R" for R(emove. The Filer will ask:

```
Remove what file ?
```

Type "TESTING.TEXT" *enter.* The display will now look like this:

```
Remove what file ? TESTING.TEXT
IBM02:TESTING.TEXT          ---> removed
Update directory ?
```

If you are really determined to get rid of TESTING.TEXT, type "Y" or "y". Any other character will leave your disk unchanged.

You can remove TESTING.CODE in the same manner. There is also a shortcut, by which you can remove both TESTING files at once. Can you discover what this is by looking at the section on "wildcards" in Chapter 3 of the Users' Guide?

At this point, it is time for you to play with the p-System on your own. Or take a break. Or, if you feel like reading some more, look at the sections in the Users' Guide that we have recommended.

Error Messages

You may, while compiling or executing your programs, encounter error messages which cause processing to halt, usually displaying an error message on the screen. There are basically four kinds of error messages which you will encounter:

Compiler Errors
These occur while you are compiling a program, and are listed in Appendix D of the *UCSD PASCAL Reference* or Appendix A of the *FORTRAN-77 Reference* manuals.

Assembler Errors

Occur when using SYSTEM.ASSMBLER and are listed in the Appendix of the *Assembler Reference* manual.

Runtime, or Execution Errors

These occur when a program is running or when using the facilities of the file handler, and are listed in Appendix A of the *User's Guide*, and Appendix A of the *FORTRAN-77 Reference.*

I/O Errors

Usually occur when a program is running, or when using the facilities of the file handler, and which are listed in Appendix B of the *User's Guide.*

Ways to configure your diskettes

The entire p-System will not fit on a single diskette (you may have guessed this from the fact that we ship it on six diskettes!). While you are using your p-System, you will want to create a set of diskettes to work with. The files on these diskettes (both p-System files and the files you create) should be arranged so that you can do your work without too much "shuffling" of diskettes in and out of the drives.

For this reason, none of your diskettes should be "packed" with files. It is often convenient to save a file on the nearest available diskette, and leaving some space on all your diskettes allows you to do this.

We recommend that you arrange your p-System in the following way:

1) A System diskette for bootstrapping.

2) A Language diskette for preparing programs.

3) A Utility diskette for frequently-used utility programs.

4) Any number of diskettes that contain your own work.

Example:

```
SYSTEM:
   SYSTEM.PASCAL
   SYSTEM.MISCINFO
   SYSTEM.INTERP
   SYSTEM.SYNTAX
   SYSTEM.LIBRARY
   SYSTEM.FILER
   SYSTEM.EDITOR
   SPOOLER.CODE

PASCAL:
   SYSTEM.COMPILER
   SYSTEM.LINKER
   LIBRARY.CODE

FORTRAN:
   SYSTEM.COMPILER
   SYSTEM.LINKER
   LIBRARY.CODE

ASSEM:
   SYSTEM.ASSMBLER
   8086.OPCODES
   8086.ERRORS
   SYSTEM.LINKER
   COMPRESS.CODE

UTILS:
   DISKFORMAT.CODE
   PRNCONFIG.CODE
   TVADJUST.CODE
   SETBAUD.CODE
   COPYDUPDIR.CODE
   MARKDUPDIR.CODE
   RECOVER.CODE
   PATCH.CODE
   DECODE.CODE
   XREF.CODE
```

The first thing to create is a diskette that will bootstrap the p-System. We often refer to this simply as the "System disk." There are certain files that MUST be on the System disk. These are:

- SYSTEM.PASCAL {the Operating System}
- SYSTEM.MISCINFO {configuration data}
- SYSTEM.INTERP {the Interpreter}
- SYSTEM.LIBRARY {the main library}

A System disk must also contain a bootstrap. The invisible bootstrap code is located at the beginning of the diskette, before all other files. It is transferred to another diskette ONLY when you transfer an ENTIRE System disk onto another diskette by using the T(ransfer command in the Filer.

Thus, if you want to create a System disk that has a different assortment of files than the System disks that we ship, T(ransfer a System disk (either SYSTEM2: or SYSTEM4:) onto a new diskette. Then R(emove those files you don't want from the new diskette, and T(ransfer other files that you do want onto the new diskette (one at a time). Filer commands are described in the Users' Guide, Chapter 3; the Filer is also discussed in Bowles's Beginners' Guide.

When you create a new System disk, remember three things. (1) You have to DISKFORMAT a brand new disk before you can use it. (2) The files listed above MUST be on the System disk. (3) SYSTEM2: is used for two-word real numbers and SYSTEM4: is used for four-word real numbers; otherwise, they contain the same information.

If the file SYSTEM.SYNTAX is on your System disk when you compile a UCSD Pascal program, the compiler will produce full error messages (when necessary). You can save some space by leaving this file off your System disk, but then the Pascal compiler will only provide error numbers, which you must look up (they are listed in an appendix to the *PASCAL Reference for the UCSD p-System.*

If you want to save six blocks on your System disk, the SYSTEM.STARTUP and SYSTEM.LOGO files can safely be removed. They are there to display the IBM logo when you bootstrap.

If you wish to use the Spooler (SPOOLER.CODE), then this file must be on the System disk. This is how SYSTEM2: and SYSTEM4: are shipped.

Once you have a System disk that suits your requirements, it is easy to go on to other things.

One "secret" to creating your set of diskettes was illustrated in the previous section. When we called the compiler by using the Operating System's R(un command, the file SYSTEM.COMPILER was on the diskette in drive #5, and NOT on the System disk.

Operating System commands like F(ile, E(dit, and C(ompile call programs that have names like SYSTEM.FILER, SYSTEM.EDITOR, and SYSTEM.COMPILER. When you use a command that requires one of these SYSTEM.xxx files, the file itself does NOT need to be on the System disk: it can be on a diskette in either drive #4 or drive #5 (except, of course, for the files already mentioned that must be on the System disk). This gives you some flexibility when you decide which files to place on a certain diskette.

With this in mind, you can create a diskette for each of the languages you plan to be using. Notice that in

the example, we have named the compiler SYSTEM.COMPILER on both the PASCAL: and the FORTRAN: diskettes. The files are different, but their names have been made the same for convenience. To compile a program in UCSD Pascal, place the diskette PASCAL: in drive #5, and use the C(ompile command or the R(un command. To compile a FORTRAN program, place FORTRAN: in drive #5, and use C(ompile or R(un in just the same way.

To run a FORTRAN program, the file SYSTEM.LIBRARY on SYSTEM: must contain certain portions of program code ("code segments"). Although the example does not show this, if you are going to use FORTRAN frequently, the SYSTEM.LIBRARY on the SYSTEM: disk will be a combination of the original SYSTEM.LIBRARY plus the code segments in FORTLIB2.CODE or FORTLIB4.CODE (whichever is appropriate to the real number size you are using). The utility LIBRARY is used to combine libraries in this way. See the FORTRAN-77 REFERENCE and the User's Guide (Chapter 5) for more details.

Remember that the System disk works for a particular size of real numbers (either two-word or four-word), and that the compilers you use (whether UCSD Pascal or FORTRAN:) must match that real number size.

Since the language diskettes as we have shown them still contain a lot of room, this is a good area to keep work-in-progress. This could consist of programs that are not yet completed. It could also consist of portions of larger programs, where the entire program is stored on some other diskette.

As you become more experienced with the p-System, you may find other arrangements that work better for your particular style of

programming, or your particular use of the IBM Personal Computer. The rule of thumb is, do what works best for you. In this section, we have made suggestions that should be generally useful. We have also pointed out some requirements of the p-System that apply regardless of what diskette configuration you eventually choose.

Large programs

This section is a little more advanced than the previous sections, and you may want to skim it on first reading. But it is probably a good idea just to read through it, in order to get an idea of some of the p-System's facilities.

As you become more experienced with the p-System, it is likely that the programs you write will become larger and more sophisticated. The larger a program is, the more cumbersome it is to keep it in a single file. On the p-System, both UCSD Pascal and FORTRAN allow the programmer to compile a program in separate portions, called "units." These units can be located in files of their own, as we will explain below.

A unit is a package of routines (for example, PROCEDUREs, FUNCTIONs, SUBROUTINEs, and so on), together with any appropriate variable and type declarations. Any number of programs may "use" a unit. Using a unit means that the program can use the unit's declarations and call the unit's routines, just as if those declarations and routines had been written into the program itself.

Much of a unit is invisible to the program that uses it. The program cannot know how the unit implements the routines that the program can use. In fact, the unit may contain other declarations that the program does not know about at all.

The advantage of this is modularity. It is possible to recompile a unit -- perhaps to fix a bug or improve an algorithm. If the unit's "interface" declarations (the ones that another program may use) are not changed, then it is NOT necessary to recompile any programs that use the unit. It should be evident that this can save a lot of time.

It is also possible for a unit to use another unit.

A codefile may contain a single program, a single unit, or a number of units. A codefile that contains a number of units is called a "library." Libraries are especially useful organizing the routines of a large program (especially one that involves several programmers), or organizing a number of general-purpose routines that many programs will use.

The library you have already encountered is SYSTEM.LIBRARY. This contains general-purpose routines for handling the IBM Personal Computer hardware. If you are using FORTRAN, SYSTEM.LIBRARY also contains routines that the p-System must use when a FORTRAN program is running.

SYSTEM.LIBRARY must reside on the System disk. But it is possible to create any number of libraries of your own, and they may reside on any diskette.

When a textfile that consists of a number of units is compiled, a library is created. It is also possible to create a library from several codefiles by using the utility LIBRARY. LIBRARY can also be used to maintain library files that have already been created.

If the unit your program uses is not in SYSTEM.LIBRARY, then the program must declare which file contains the unit. This filename must also appear in a file called a "library text file." This is simply a textfile that contains names of user's library

files. The default library text file is called USERLIB.TEXT, and must reside on the System disk.

When a program is run, the p-System searches for each unit that the program uses. First it searches the files named in USERLIB.TEXT, and then the units in SYSTEM.LIBRARY. If it cannot find the unit at all, it gives an error message.

It is actually possible to have more than one library text file. When you eX(ecute a program, you can specify which library text file the p-System is to use (this is done with an "execution option string": see Chapter 2 in the Users' Guide). Since the files named in a library text file are searched in the order they appear, it is possible to arrange different library text files that are more efficient for running certain programs.

All these facilities make it possible to break a large program into smaller pieces. Many of the pieces can be made general-purpose, which saves much effort when you are developing programs at a later time. By breaking down a large program in this way, you can make it easier to debug and faster to compile, and avoid running out of space on your diskettes.

For more information on these topics, see Chapter 5 in the Users' Guide. Units in UCSD Pascal are described in the *PASCAL Reference*, and units in FORTRAN are described in the *FORTRAN-77 Reference* manual.

NOTES

CHAPTER 4. MAKING USE OF THE IBM PERSONAL COMPUTER HARDWARE

Contents

The Display 4-3
The Printer 4-4
Remote Devices 4-5
Extended Memory 4-6

NOTES

This chapter talks about some p-System utilities that allow you to control the IBM Personal Computer hardware. The utilities mentioned here are described in Chapter 7 of the *User's Guide*. Most of them are shipped on the UTILITY: diskette.

The Display

If the display you use for your IBM Personal Computer is a television set with an RF modulator, then you may need to adjust it whenever you bootstrap the p-System. This is done with the utility TVADJUST.

You will need to adjust your TV set if, when you bootstrap, there are one to three columns of characters MISSING on the left-hand side of your screen. To fix this, put the UTILITY: diskette in drive #5. Type "X" for eX(ecute, and when the p-System says:

Execute what file ?

... type "#5:TVADJUST" and press *enter*. You will see a scale on the screen. Type the *right-arrow* key until the first column of text is visible on the left. If you go too far, use the *left-arrow* key to adjust in the other direction. When the display is properly centered, type *enter*.

If the display is adjusted too far to the left, the TV's timing synchronization may be lost, and the image will start to "roll". Don't panic, simply adjust the display back to the right with the *right-arrow* key.

The display will function properly until you turn the IBM Personal Computer off or bootstrap again. You will need to use TVADJUST every time that you bootstrap the p-System.

The file SYSTEM.MISCINFO contains information about the IBM Personal Computer, the display, and the use of various characters on the keyboard. It is possible to alter SYSTEM.MISCINFO by using the utility SETUP.

It is possible to use SETUP to alter the data items in SYSTEM.MISCINFO that describe the keyboard configuration, and so forth. However, we do not anticipate that you will need to do so, since the SYSTEM.MISCINFO file that is shipped with the p-System is tailored to the IBM Personal Computer and its keyboard.

The Printer

There are a number of ways to use the IBM Personal Computer printer from the p-System.

The printer is a standard p-System device called PRINTER: or device #6. Programs may write to this device. A textfile may be printed by simply entering the Filer and using T(ransfer to send the file to PRINTER:

The SPOOLER utility allows the user to create a "queue" of files to be printed, and print them concurrently with other p-System activities. For example, the user selects three files to print, eX(ecutes SPOOLER, and places the three filenames in the queue. When the user Q(uit's SPOOLER, it begins printing the files. In the meantime, the user may be editing a fourth file using the p-System Editor.

> **Note:** To use the SPOOLER, you must first eX(ecute the "SETUP utility". See "SETUP Utility" in Chapter 7 and set "HAS SPOOLING" to true, then replace your SYSTEM.MISCINFO file as directed.

Because SPOOLER is frequently useful, it is shipped on the diskettes SYSTEM2: and SYSTEM4:.

When you bootstrap your p-System, the printer is set to type 80 characters per line (like the display). When you make listings of compiled programs, you should have the printer type 132 characters per line (if you do not do this, you will get some overprinting). This can be done by using the utility PRNCONFIG. PRNCONFIG may also be used to adjust other printer settings, such as tab stops or the number of lines per inch.

If you use PRNCONFIG, you must eX(ecute it every time that you bootstrap the p-System. An alternative method of changing the printer to 132-character lines is to send it the character ASCII 15. This can be done from within a program, or by creating a file that contains this character, and T(ransfer'ring it to the printer.

> **Note:** If you intend to use the printer, it must be turned on when you bootstrap the system. Otherwise, it will not appear in the p-System's inventory of volumes on line, and you will get an error message when you try to use it.

Remote Devices

The p-System device ports called REMIN: and REMOUT: may be connected with remote devices, or the same remote device (such as a Modem). These are connected via the standard RS-232 adapter available for the IBM Personal Computer, and connect through that part to devices that the user may supply. The baud rate of these ports may be set by the utility SETBAUD, in order to adjust to the requirements of different remote hardware.

Making Use of IBM Personal Computer Hardware

Extended Memory

If you have installed the optional 32K or 64K Memory Expansion options in your IBM Personal Computer, you will want to take advantage of that additional memory (you will notice a speed advantage in running certain programs, for example, the SYSTEM.COMPILER).

In order to reconfigure your system in this manner, you will need to X(ecute the utility SETUP which is included on your UTILITY: diskette. You should read through the discussion of the setup utility in Chapter 7 of your *User's Guide for the UCSD p-System.*

If you have installed the 64K Memory Expansion option, you should C(hange the following fields:

- Set the field 'code pool base[first word]' to 1 (default is 0).
- Set the field 'has extended memory' to true (default is false).

If you have installed the 32K Memory Expansion option, you should C(hange the following fields:

- Set the field 'code pool size' to H3FFF (the default value of this field is H4FFF - the field is ignored unless has extended memory is true.
- Set the field 'has extended memory' to true.

After you have Q(uit the SETUP program, make sure that you update the SYSTEM.MISCINFO file by changing the name of the NEW.MISCINFO to SYSTEM.MISCINFO, to record the changes you have made.

Dual-Sided Diskette Drives

The p-System takes advantage of all of the storage space available on a dual-sided diskette. If you have dual-sided diskette drives installed in your IBM Personal Computer, you should proceed as follows:

Formatting Your Diskettes

The DISKFORMAT utility which was outlined in this manual automatically formats both sides of your diskette.

If you have a dual-sided diskette drive in your system, you will see the following message when the program has completed formatting the diskette:

640 block disk formatted

Creating System Diskettes

To create a dual-sided SYSTEM2: or SYSTEM4: diskette, you should first use the DISKFORMAT utility to format a new diskette, and then follow the instructions for creating a backup diskette, answering "Y" to the prompt:

Transfer 320 blocks ? (Y/N)

After you have successfully transferred the p-System to the dual-sided diskette, Q(uit the Filer, remove the new system diskette from drive #5 and insert the UTILITY: diskette into drive #5.

You can now eX(ecute a utility program called DISKSIZE, that changes the information contained in the diskette directory that records the size of the diskette (in blocks).

To do this, from the system level, type "X" and when you see the prompt:

Execute what file?

Type "#5:DISKSIZE" and press the Enter key.

DISKSIZE should respond as follows:

DISK SIZE CHANGER [A2]

Change directory size on what unit? (4,5)

At this point, insert the SYSTEM diskette into drive #4 with the newly created SYSTEM diskette.

Now type "4" and press Enter.

DISKSIZE should then ask:

What is the new directory size in 512 byte blocks?

You should type in "640" and press Enter.

The program changes the size information contained in the directory of the new disk. When it is finished, it will prompt:

Insert system disk and press enter

Insert the SYSTEM diskette into drive #4 (if it is not already there) and press Enter. You now have a fully formatted dual-sided diskette.

Creating Storage Diskettes

The procedure just described can also be used for backing up the diskettes that were shipped with the p-System for use on dual-sided disk drives (it must be used for system diskettes, since the bootstrap record should be transferred by T(ransferring an entire diskette.

To reconfigure your compiler diskette, or create storage diskettes for your files, you should proceed as follows:

1. Use DISKFORMAT to format a new diskette.

2. Enter the filer and type "Z" for Z(ero, to set up a directory on that volume (see the *User's Guide for the UCSD p-System* for information on the Z(ero command). When the program prompts:

   # of blocks on the disk ?

3. Type "640" and press Enter.

4. Continue to respond to the prompts as instructed in the *User's Guide for the UCSD p-System.*

You now have a fully formatted dual-sided diskette. If you ask for a L(isting of the directory on that diskette, the number of unused blocks will be based on a total of 640.

> **Note:** Remember that you are still limited to a total of 77 files on a diskette.

Serial Printer Support

The UCSD p-System for the IBM Personal Computer is configured so that output directed to the device PRINTER: (#6:) is sent to the Parallel Printer Adapter. Output to the device REMOUT: (#8) is sent to the Asynchronous Communications Adapter (serial interface). The default configuration for the REMIN: and REMOUT: ports is as follows:

- Baud rate = 300
- Parity = none
- Number of stop bits = 1
- Word length = 8

If you have a modem (modulator/demodulator), or another communication device connected to the Asynchronous Communications Adapter, these parameters normally do not need to be changed, with the exception of the baud rate, which can be changed by eX(ecuting the SETBAUD utility (see "SETBAUD Utility" in Chapter 7.)

If you have a serial printer connected to the Asynchronous Communications Adapter, and wish to send output to that device, you have two options:

1. You can send your output to the serial printer via the REMOUT: device, using the RS232SET utility (shipped on the EXTRAS: diskette) to configure the port for your printer.

2. You can reconfigure your system diskette so that output sent to the device PRINTER: will automatically be routed through the Asynchronous Communications Adapter to your serial printer. You will need to eX(ecute the RS232SET utility to configure the system to your printer specifications.

The remainder of this section describes the procedures for configuring your system to either of these options. To use a serial printer with the p-System, you must follow these steps:

1. Connect your printer to the Asynchronous Communications Adapter.

2. Configure a system diskette to support your serial printer.

3. Use the RS232SET utility to configure the serial interface to the parameters that match your printer.

Connecting Your Printer

Before you start, we suggest that you read the manual that accompanied your printer to familiarize yourself with the printer's configuration. Note that the printer must have a standard RS232 interface and that the Clear To Send (CTS) line must indicate that the printer can accept a character to print. If the printer cannot accept a character to print (buffer full, etc.), it must not assert the CTS signal. If your serial printer connection does not match this description, you must modify the pin connections accordingly, or the printer will not work correctly. (Consult your IBM Personal Computer Dealer for additional information.)

Reconfiguring Your System Diskette

The following files, included on your EXTRAS: diskette, are needed to reconfigure your system for a serial printer:

- INTERPX.2.CODE or INTERPX.4.CODE
- RSP.CODE
- BIOS.CODE or BIOS.S.CODE
- TERTBOOT.CODE
- RS232SET.CODE
- SYSTEM.LINKER
- COMPRESS.CODE

To avoid running out of space on your diskette, T(ransfer all of these files to a scratch diskette, so that you will have enough space left on the diskette to do the reconfiguration. Notice that you must make two file selections based on your individual requirements: you must first decide whether you want to configure your diskette for 2 or 4 word real numbers. Use INTERPX.4.CODE for four word real numbers, and INTERPX.2.CODE for two word real numbers. Remember that you must use the corresponding SYSTEM.PASCAL, SYSTEM.COMPILER and SYSTEM.LIBRARY. The choice between BIOS.S.CODE and BIOS.CODE should be made as follows:

- BIOS.S.CODE should be used if you want to redirect the output for device PRINTER: to the Asynchronous Communications Adapter.
- BIOS.CODE should be used if you want to continue to use PRINTER: to refer to the Parallel Printer Adapter, but wish to attach a serial printer to your REMOUT: port (the Asynchronous Communications Adapter).

To reconfigure your diskette, proceed as follows:

Insert your scratch diskette containing the files previously listed into drive #5:. Enter the F(iler and change the default Prefix to #5: by typing "P" and then answering #5: to the prefix prompt. Q(uit the F(iler and type "L" from the system level to call the L(inker.

The L(inker will respond with a series of questions. Your responses are noted on the right below:

Linker IV.0 [x7]

Program prompt:	User Response:
Host file?	INTERPX.4 (or
Opening INTERPX.4.CODE	INTERPX.2)
Lib file?	RSP
Opening RSP.CODE	
Lib file?	BIOS.S (or BIOS)
Opening BIOS.S.CODE	
Lib file?	TERTBOOT
Opening TERTBOOT.CODE	
Lib file?	<ENTER>
Map name?	<ENTER>
Reading INTERP86	
Reading RSP	
Reading BIOS.S	
Reading TERTBOOT	
Output file?	LINKER.OUT
Linking INTERP86 # 1	
Copying proc INTERP86	
Copying proc RSP	
Copying BIOS.S	
Copying TERTBOOT	

You should then see the system promptline at the top of your screen. Now press "X" and when you see:

Execute what file?

Type COMPRESS. COMPRESS prompts with a series of questions to which you should respond as follows:

Assembly Code File Compressor IV.0 [g4]

Type "!" to escape

Do you wish to generate relocatable code file: N
Base address of relocation (hex): O
File to compress: LINKER.OUT
Output file (<ent> for same): SYSTEM.INTERP
Procedure #0: 0000H – 21B5H 8630 bytes
Procedure #1: 21B6H – 2521H 876 bytes
Procedure #2: 2522H – 329FH 3454 bytes
Procedure #3: 32A0H – 35EBH 844 bytes

Highest code address is 35EBH.
Output file is 13804 bytes long.

> Note: Prompts are in green. Your responses are indicated after the prompts.

After this operation is completed, return to the Filer and T(ransfer the newly created SYSTEM.INTERP file to your SYSTEM diskette, to replace the old SYSTEM.INTERP file. Make sure that, if you have used the INTERPX.2.CODE, you replace the SYSTEM.INTERP file on your SYSTEM2: diskette and, if you need the INTERPX.4.CODE, you T(ransfer it to the SYSTEM4: diskette. An attempt to use floating-point numbers of mismatched size will result in the termination of the program.

If you have two Asynchronous Communications Adapters, you will need to make some modifications to the second card. If you are unfamiliar with how to do this, you should consult your authorized IBM Personal Computer dealer.

The RS232SET utility refers to the serial interfaces as 0 and 1. 0 is the primary card, and 1 is the secondary (modified) card. Each can be configured separately. Output to the serial port always defaults to the primary interface. If you wish to direct output to the secondary serial interface from within a program, you must use the Select_Remote procedure in the IBMSPECIAL Unit (parameter to the procedure is either 0 or 1). (Similarly, the Select_Printer procedure in the IBMSPECIAL unit directs output to a secondary parallel interface.)

RS232SET Utility

You are now ready to configure the serial interface to the parameters required by your printer, using the RS232SET utility supplied on your EXTRAS: diskette. This information should be contained in the manual that accompanied your printer. Make sure that the SYSTEM diskette in drive #4 contains the SYSTEM.INTERP file configured for serial printers which you just created. Insert your scratch diskette containing the RS232SET utility into drive #5. Press 'X' from the system promptline. You should see:

Execute what file?

Type #5:RS232SET and press Enter. RS232SET should display:

```
RS232SET: B(aud rate, S(top bits, P(arity,
          W(ord length, Q(uit, ? [a.1]

Current RS232 configuration (port 0) is:
     baud rate = 300
     parity = none
     number of stop bits = 1
     word length = 8
```

The B(aud rate, S(top bits, P(arity and W(ord length commands are used to change the configuration. Commands are listed below, together with the menus that are presented to you when you invoke them.

B(aud rate

Type "B" from the initial promptline. The program should display:

```
Baud rate choices:
  A)  110 baud
  B)  150 baud
  C)  300 baud
  D)  600 baud
  E)  1200 baud
  F)  2400 baud
  G)  4800 baud
  H)  9600 baud

Enter baud rate choice:
```

Type the letter corresponding to the baud rate required by your printer.

S(top bits

Type "S" from the initial promptline. You should see:

Stop bit choices:
A) 1 stop bit
B) 2 stop bits

Enter stop bit choice:

Type the letter corresponding to the number of stop bits required by your printer.

P(arity

Type "P" from the initial promptline. You should see:

Parity choices:
A) no parity
B) odd parity
C) even parity

Enter parity choice:

Type the letter corresponding to the appropriate choice.

T(oggle port

> **Note:** T(oggle port is not shown on the initial promptline.

This command toggles between serial port 0 and serial port 1, if you have two Asynchronous Communication Adapters installed and wish to configure both of them.

When you execute the RS232SET utility, the default configuration of the primary interface (port 0) is displayed. Pressing "T" will display the current configuration of the secondary interface (port 1).

You can then proceed to reconfigure the other port with the B(aud rate, S(top bits, P(arity and W(ord length commands.

D(efault

> **Note:** D(efault is not shown on the initial promptline.

Ensure that the SYSTEM diskette that you have configured for serial printers is in drive #4. Type "D" from the initial promptline.

The D(efault command permanently changes the configuration of the serial port to conform to the parameters that you have established with the RS232SET utility. Once the D(efault command has been completed, the displayed configuration will be the default at subsequent bootstrapping of the system.

The D(efault command only changes the permanent configuration of the serial port displayed (0 or 1). To change the default configuration of the other serial port, you must first use the T(oggle command to switch to the other port, and then press "D" again.

APPENDIX A. THE p-SYSTEM PACKAGE

Contents

Documents shipped with the p-System A-3
Diskettes and files shipped with the p-System A-4

NOTES

Documents shipped with the p-System

- *Operations Guide for the UCSD p-System* (Part 1)
- *User's Guide for the UCSD p-System* (Part 2).
- *Beginner's Guide for the UCSD p-System.*
- *Internal Architecture for the UCSD p-System.*
- *Assembler Reference for the UCSD p-System.*
- *UCSD PASCAL Reference for the UCSD p-System* (shipped only with UCSD Pascal).
- *FORTRAN-77 Reference for the UCSD p-System* (shipped only with FORTRAN-77).

Diskettes and files shipped with the p-System

STARTUP:

SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.INTERP
SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.STARTUP
SYSTEM.SYNTAX
NAMEFILE
SCDEMO.CODE
COPYSCUNIT.CODE
UPDATE.CODE
COMPDEMO.TEXT
EDITDEMO.TEXT
UPDATE.TEXT

SYSTEM2:

SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.INTERP
SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.STARTUP
SYSTEM.LOGO
SYSTEM.SYNTAX
SYSTEM.LIBRARY
SPOOLER.CODE

SYSTEM4:

{files are the same as SYSTEM2:, but the System defaults to 4-word real numbers}

UTILITY:

- XREF.CODE
- TAPE.CODE
- CODEGEN.CODE
- LIBRARY.CODE
- PATCH.CODE
- DECODE.CODE
- SETUP.CODE
- COPYDUPDIR.CODE
- MARKDUPDIR.CODE
- RECOVER.CODE
- PRNCONFIG.CODE
- TVADJUST.CODE
- SETBAUD.CODE
- DISKFORMAT.CODE
- DISKSIZE.CODE

EXTRAS:

- SYSTEM.LINKER
- SYSTEM.ASSMBLER
- 8086.OPCODES
- 8086.ERRORS
- COMPRESS.CODE
- GOTOXY.CODE
- GOTOXY.TEXT
- SPOOLOPS.CODE
- COMMANDIO.CODE
- KERNEL.CODE
- SCREENOPS.CODE
- 8087.FOPS
- INTERPX.2.CODE
- INTERPX.4.CODE
- RSP.CODE
- BIOS.CODE
- BIOS.S.CODE
- TERTBOOT.CODE
- RS232SET.CODE

PASCAL: (shipped only with UCSD Pascal)

SYSTEM.COMPILER
PASCAL4.COMPILE

FORTRAN: (shipped only with FORTRAN-77)

FORTRAN2.CODE
FORTLIB2.CODE
FORTRAN4.CODE
FORTLIB4.CODE

Personal Computer
Computer Language
Series

USERS' GUIDE

for the UCSD p-System™ Version IV.0

Produced by SofTech Microsystems, Inc.
Edited by Keith Shillington, Gillian Ackland,
Randy Clark and Stan Stringfellow

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1982
© Copyright Regents of the University of California 1978
© Copyright SofTech Microsystems, Inc. 1979, 1980, 1981

UCSD, UCSD Pascal, and UCSD p-System are all trademarks of the Regents of the University of California.

CONTENTS

BOOK 2 USER'S GUIDE

CHAPTER 1. INTRODUCTION 1-1
How to Use this Manual 1-3
Overview 1-5
System Rationale and Organization 1-5
File Organization 1-9
Device and Volume Organization . 1-14
Program and Library Organization 1-18
Note on Bug Reporting 1-23

CHAPTER 2. THE SYSTEM COMMANDS 2-1
Booting and Promptlines 2-3
Diskette Swapping 2-6
Execution Option Strings 2-7
Alternate Prefixes and Libraries ... 2-8
Redirection 2-10
Function Redirect 2-13
Procedure Exception 2-14
Procedure Chain 2-14
Individual Commands Alphabetically ... 2-16
Prompts for Filenames 2-16
ASSEMBLE 2-17
COMPILE 2-18

EDIT 2-19
FILE 2-19
HALT 2-20
INITIALIZE 2-21
LINK 2-21
MONITOR 2-22
RUN 2-24
USER RESTART 2-24
EXECUTE 2-25

CHAPTER 3. FILES AND FILE HANDLING 3-1
Types of Files 3-3
File Formats 3-3
Volumes 3-5
The Workfile 3-6
Filenames 3-6
Using the Filer 3-8
Prompts in the Filer 3-8
Names of Files 3-10
Filer Commands 3-12
B(ad blocks 3-13
C(hange 3-14
D(ate 3-17
E(xtended list 3-18
G(et 3-20
K(runch 3-21
L(ist directory 3-22
M(ake 3-26
N(ew 3-27
P(refix 3-28
Q(uit 3-28
R(emove 3-29
S(ave 3-31
T(ransfer 3-32
V(olumes 3-38
W(hat 3-38
eX(amine 3-39
Z(ero 3-40
Recovering Lost Files 3-42
Lost Directories 3-44

CHAPTER 4. THE SCREEN ORIENTED EDITOR 4-1
Introduction 4-3
The Concept of a Window into the File 4-3
The Cursor 4-4
The Promptline 4-4
Notation Conventions 4-5
The Editing Environment Options 4-5
Getting Started 4-5
Entering the Workfile and Getting a Program 4-6
Moving the Cursor 4-7
Using Insert 4-8
Using Delete 4-9
Leaving the Editor and Updating the Workfile 4-10
Using the Editor 4-11
Command Hierarchy 4-11
Repeat Factors 4-12
The Cursor 4-12
Direction 4-12
Moving the Cursor 4-13
Entering Strings in F(ind and R(eplace 4-15
Screen Oriented Editor Commands 4-16
A(djust 4-16
C(opy 4-18
D(elete 4-20
F(ind 4-22
I(nsert 4-25
J(ump 4-29
K(olumn 4-30
M(argin 4-31
P(age 4-33
Q(uit 4-33
R(eplace 4-36
S(et 4-38
V(erify 4-44
eX(change 4-44
Z(ap 4-46

CHAPTER 5. SEGMENTS, UNITS, AND LINKING 5-1
Overview 5-5
Main Memory Management 5-5
Separate Compilation 5-6
General Tactics 5-8
Segments 5-11
Units 5-13
The Linker 5-18
Using the Linker 5-19
The Utility LIBRARY 5-21
Using LIBRARY 5-23
SYSTEM.LIBRARY Routines 5-26
Screen Control Unit 5-26
Unit Commandio 5-33
Unit IBMSPECIAL 5-33
Turtlegraphics 5-40

CHAPTER 6. CONCURRENT PROCESSES 6-1
Introduction 6-3
Semaphores 6-6
Mutual Exclusion 6-8
Synchronization 6-9
Other Features 6-11

CHAPTER 7. UTILITIES 7-1
Preparing Assembly Codefiles 7-5
Preparing Codefiles for Compression 7-6
Running COMPRESSOR 7-7
Action and Output Specification ... 7-8
PATCH 7-10
EDIT Mode 7-10
TYPE Mode 7-12
DUMP Mode 7-13
A Note on Prompts 7-16
DECODER 7-16
Duplicating Directories 7-22
COPYDUPDIR 7-22
MARKDUPDIR 7-23

Procedural Cross-Referencer -- XREF .. 7-24
Introduction 7-24
Referencer's Output 7-25
Using Referencer 7-28
Limitations 7-30
The Debugger 7-31
Invoking and Exiting the
Debugger 7-33
Displaying and Altering Memory ... 7-35
Further Single-Stepping Options... 7-36
Example of Debugger Usage 7-38
Summary of the Commands 7-39
The RECOVER Utility 7-41
The TAPE Utility 7-43
The Print Spooler 7-44
The Native Code Generator 7-45
The TV Adjust Utility 7-48
the SETBAUD Utility 7-49
The Printer Configuration Utility 7-49
The Disk Format Utility 7-50
SETUP 7-51

APPENDIX A. EXECUTION ERRORS.... A-1

APPENDIX B. I/O RESULTS B-1

APPENDIX C. DEVICE NUMBERS C-1

APPENDIX D. AMERICAN STANDARD
CODE FOR INFORMATION
INTERCHANGE (ASCII) D-1

**APPENDIX E. SPECIAL KEYS ON THE
IBM PERSONAL COMPUTER............** E-1

NOTES

CHAPTER 1. INTRODUCTION

Contents

How to use this Manual 1-3
Overview 1-5
p-System Rationale and
Organization 1-5
File Organization 1-9
Device and Volume Organization 1-14
Program and Codefile Organization ... 1-18
Note on Bug Reporting 1-23

NOTES

How to use this Manual

This is the basic reference for the UCSD p-System on the IBM Personal Computer. It should contain answers to your questions concerning the p-System but it is not meant to be a tutorial about the use of the p-System. If you have never used the UCSD p-System before, you should consult Ken Bowles' *Beginner's Guide for the UCSD p-System.* For a description of UCSD Pascal see the *PASCAL Reference for the UCSD p-System* by Clark and Koehler.

Although this Manual is not a tutorial, this introduction is designed as a description of the overall structure of the p-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 the Text Editor and file-handling, then you should approach this manual. Booting is explained in Chapter 2, and the IBM Personal Computer Keyboard is described in Appendix E. The chapters on the Filer and the Screen Oriented Editor will prove useful. If you intend to work with large programs, you should definitely read Chapter 5: Segments, Units, and Linking.

Much of the manual -- for example, sections on utilities, and the appendixes -- 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.

FORTRAN is described in the *FORTRAN-77 Reference for the UCSD p-System.*

The 8086/88/87 Assembler is described in the *Assembler Reference for the UCSD p-System.*

Details concerning the internal workings of the p-System are discussed in the Internal Architecture Guide.

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 p-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 the p-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 the environment, which will allow you to develop your own ways of using the p-System. In time you will learn to use subtler forms of the commands, and develop your own shortcuts.

We hope that you are creative in your use of the p-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 (<enter> or <escape>), and names of things within a literal description are represented this way as well:

IF <Boolean expression> THEN <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

Overview

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

p-System Rationale and Organization

The UCSD p-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 Operating System, Filer, and Editors 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 manual, particularly Chapter 2, 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 p-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 p-System by being available for continual monitoring and control of running programs and I/O devices.

The p-System runs on top of an 8086/88/87 assembly language Interpreter which executes *all* programs written in high-level language. While the Assembler generates machine language for the 8086/88 and 8087 processors, the p-System's high-level languages, Pascal and FORTRAN, are compiled to an intermediate language called P-code. This code is in the form of machine code for an idealized "P-machine".

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

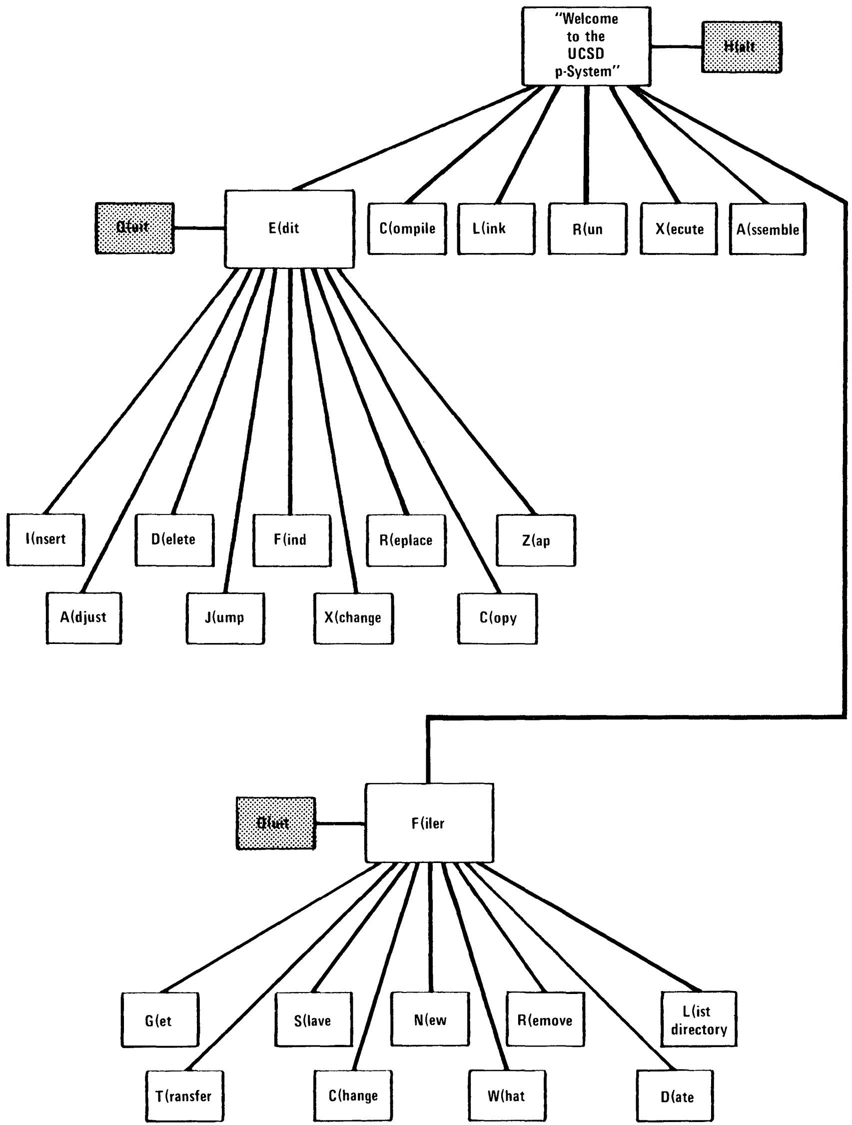


Figure 1. Command structure.

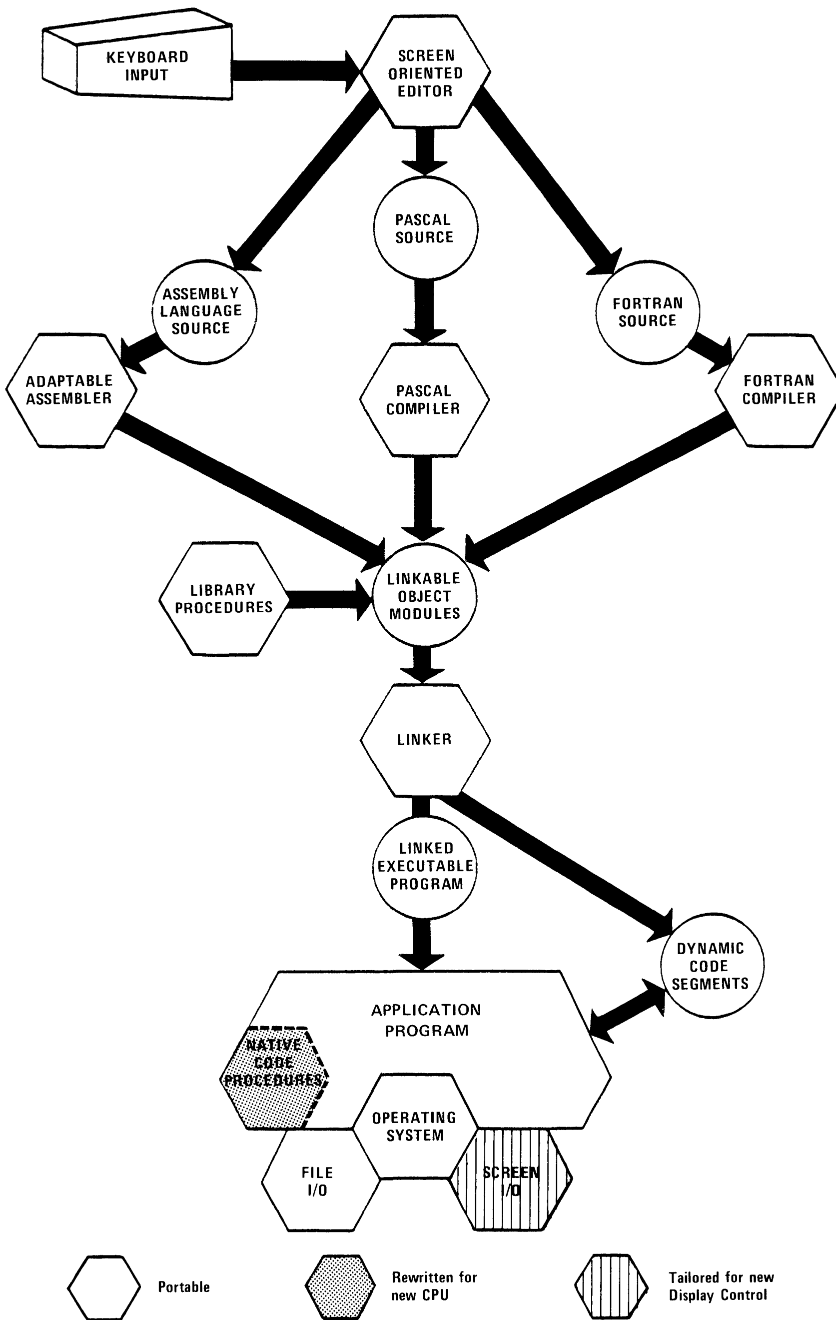


Figure 2. Interrelationships among components.

For more detailed information on the various p-System commands, refer to Chapter 2.

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

File Organization

A file, to the UCSD p-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 p-System will treat certain files in very specific ways, and there are naming conventions which support this special treatment. The naming conventions inform the p-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 floppy disks. Each such disk contains a directory which describes up to 77 files.

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, the printer, the CRT, and so forth). It also provides for some management of storage units themselves. More information on this is provided in "Device and Volume

Organization" in this Chapter, and the Filer is described thoroughly in Chapter 3.

> **Note:** Bootstrapping the p-System involves reading files off of a particular disk. That disk is called the "System disk", "default disk", or "bootstrap disk." In the p-System's syntax for filenames, it is called *, and when a file name is shown preceded by a star (for example, *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 3.

System Files

The files which comprise the major portions of the p-System itself are identified by the prefix SYSTEM.. Thus, important files are SYSTEM.PASCAL, SYSTEM.EDITOR, SYSTEM.ASSMBLER, and so forth. This section gives a general description of the names of the major pieces of the p-System.

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

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

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

SYSTEM.COMPILER is not necessarily Pascal -- it could be the FORTRAN compiler. In this way, by changing the appropriate file to SYSTEM.COMPILER, a user may invoke that compiler with a single keystroke.

SYSTEM.SYNTAX

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

SYSTEM.LIBRARY

... contains previously compiled or assembled routines to be linked into, libraried into or used by other programs.

SYSTEM.STARTUP

... is an executable codefile. If a file with this name exists when the System is bootstrapped or I(nitialize'd, that file is executed before the main System promptline is displayed.

SYSTEM.MISCINFO

... is a data file containing miscellaneous data items pertaining specifically to the IBM Personal Computer hardware -- most of it is devoted to terminal-handling information.

SYSTEM.INTERP

... is the P-machine emulator on which the rest of the system runs.

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 "Workfile" in this chapter. Before discussing workfiles, we will talk about more ordinary user files.

When the p-System is bootstrapped, certain System files must be on the disk it is bootstrapped from. These are SYSTEM.PASCAL, SYSTEM.INTERP, and SYSTEM.MISCINFO. Also, SYSTEM.LIBRARY whenever it is to be used, must be on the boot disk unless a library text file specifies a SYSTEM.LIBRARY on another disk (see Chapter 5 for a description of library text files). Other System files may be anywhere. The p-System will search for them whenever it is bootstrapped or I(nitialized (see Chapter 2). Whenever it needs them and they are not on the device where it previously found them, it will again search for them. First the p-System will search the System disk, and then any other disks that are on-line.

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, GAME.TEXT, or even SYSTEM.WRK.TEXT, are human-readable files, formatted for use by the p-System's Editor. They include a header block, and follow certain internal conventions.

.CODE files, such as SORTER.CODE, GAME.CODE, or SYSTEM.WRK.CODE, are either P-code or native code. P-code is the code generated by the p-System's compilers and executed on the P-machine Interpreter. Native code refers to assembly language code that is ready to run on the 8086/88 processor. .CODE files are typically the output of a compiler or an assembler; they may also

be generated by the Linker or Library from a group of previously existing codefiles.

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

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

These naming conventions *do* matter to certain other System programs -- for example, the Editor will only edit .TEXT files. A codefile must be created with the .CODE suffix; once it is created, the name can be changed to something else, and it will still be executable. 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 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.WKR.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, which is the default name for a compiled listing, 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 File 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 p-System commands and the Filer, and of course, playing with the p-System yourself.

Device and Volume Organization

The various peripherals that the p-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 p-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 p-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 : (for example, 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 give 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 p-System is booted it is *, the System disk) so that the user need not type out the prefix every time a file is needed.

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

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

Device Number	Volume Name	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTERM:	screen and keyboard without echo
4	<disk name>:	the system disk
5	<disk name>:	the alternate disk
6	PRINTER:	a parallel printer output line
7	REMIN:	a serial input line
8	REMOUT:	a serial output line
9..12	<disk name>:	additional drives

This table is given, with some further exposition, in Chapter 3 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 p-System's treatment of devices. Most use of the p-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 p-System accomplishes device control through a portion of the Interpreter called the BIOS (for Basic I/O Subsystem). The BIOS contains the device drivers.

The p-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. Generally, users of the IBM Personal Computer should have no need to alter SYSTEM.MISCINFO or GOTOXY, however.

The p-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 p-System. The Filer can communicate with other devices, and so can a user's program (either using a language's standard I/O routines, or using special p-System intrinsics which can be much more efficient).

It is also possible to temporarily redirect the input or output of a program or the p-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 p-System itself to be driven by script files (a useful test tool). Refer to the eX(ecute and M(onitor commands in Chapter 2, and the UCSD intrinsics REDIRECT, EXCEPTION, and CHAIN also described in Chapter 2.

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, it is frequently necessary to break a program up and compile it in two or more pieces.

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

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

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

This section and the rest of the manual use the term "routine" to mean a procedure, function, or process, and the term *compilation unit* to refer to a program or

UNIT. A compilation unit which uses separately compiled routines is called a "host compilation" or "client".

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

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

In the p-System, a codefile is organized into "segments". A compilation unit contains at least one segment -- the routines and data of the compilation unit itself. This segment is called the "principal segment". If the compilation unit contains SEGMENT routines (see below), each segment routine will be a "subsidiary segment" that accompanies the principal segment. If the compilation unit references separately compiled UNITs, those are *not* considered subsidiary segments, but *are* named in a list of segment references that accompanies the principal segment. Segments are the basic unit of transfer when code is read from a disk or removed from memory.

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

When a host program that uses other units is executed, the p-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 p-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 2). A compilation unit can also specify the library it needs by using the $U Compiler option (see PASCAL REFERENCE for the UCSD p-System). Libraries are discussed in detail in Chapter 5.

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 SYNCHPROCS:

      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, for example:

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

The assembled routines must carefully adhere to Pascal's calling and parameter-passing conventions, and respect p-System constraints on the use of machine resources such as registers. See *Assembler Reference for the UCSD p-System.*

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. 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 p-System to better allocate memory, since only those segments that are being used need to 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 *PASCAL Reference for the UCSD p-System*, and the *Internal Architecture Guide for the UCSD p-System*).

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 2.

Using the p-System, standalone assembly language programs can be created, linked, loaded, and run. See the Assembler Reference and Chapter 7 on the COMPRESSOR utility.

A fuller discussion of the questions of separate compilation, linking, and memory management, is given in Chapter 5.

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. 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.

What is required in a bug report? A phone call, or letter, or a Problem Report mailed to your dealer, may all be adequate, but only if they contain certain information. 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 do not claim that your suggestion reports a bug -- that only confuses the issue. Your standard here is the Users' Guide. 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 the p-System 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.

NOTES

CHAPTER 2. THE SYSTEM COMMANDS

Contents

Booting and Promptlines 2-3
Diskette Swapping 2-6
Execution Option Strings 2-7
Alternate Prefixes and Libraries 2-8
Redirection 2-10
Function REDIRECT 2-13
Procedure EXCEPTION 2-14
Procedure CHAIN 2-14
Individual Commands Alphabetically ... 2-16
Prompts for Filenames 2-16

NOTES

This chapter describes the bootstrapping process for the UCSD p-System on the IBM Personal Computer. It 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 p-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 p-System, which indeed it is -- you have already (in Figure 1) seen the p-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 submodules. 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 p-System was from the start conceived as an interactive environment. The point is that the p-System commands are what you must use to accomplish interaction between the various p-System components.

Booting and Promptlines

In order to boot the UCSD p-System on the IBM Personal Computer position the system unit switch to off. Insert either of the system diskettes (SYSTEM2 or SYSTEM4), label up, into the left disk drive (which is referred to as #4: or drive A). Position the system unit switch to on and the p-System will boot automatically. In order to re-boot, you may

either turn the power off and on again or, while holding down the two keys Ctrl and ALT, press the DEL key on the number pad. When the booting process has completed, the IBM Personal Computer display will appear along with the Command: promptline of the UCSD p-System.

A promptline (sometimes called a menu) shows the command options at any given level of the p-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 p-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.02 B3i-D]

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]

>**Edit: A(djst C(py D(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch Z(ap [E.7n]**

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: Q(uit, B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(ols,? [C.11]

... is the remainder of the Filer's menu.

SYS COMMANDS

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 <enter>

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 p-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 p-System commands.

Diskette Swapping

Since the IV.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 diskettes at various times (especially while running the p-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 SEGNAME not found on device #5
Put volume USER1 in unit 4
type <space> to continue

... in this example, the p-System requests the disk USER1:, and will wait until the user types *space*. (If the user types *space* but has not replaced USER1:, the p-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 p-System did not expect, the p-System considers the 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 p-System displays a prompt of the following sort:

Please replace volume USER2 in device #5
Type <space> to continue

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

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

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

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 3 and Chapter 1), and changing the default library text file (see Chapter 1 or Chapter 5). 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 (Chapter 2) and possibly the EXCEPTION intrinsic (Chapter 2). The intrinsic CHAIN (Chapter 2) 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 Chapter 2.

> **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.

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 3).

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 5.

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).

Redirection

The following execution option strings control redirection:

PI=*filename*

PI=*string*

PO=*filename*

I=*filename*

I=*string*

O=*filename*

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

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

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

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

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

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

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

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

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

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 not device-level intrinsics such as UNITREAD, UNITWRITE, BLOCKREAD, BLOCKWRITE, and so on. It also cannot affect calls of the form:

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

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

Examples: **Execute what file? YEEN PI=IN PO=OUT**

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

Execute what file? I=

... stops System input redirection.

Execute what file? I="fgRUNME,qr"

... this would:

```
f: enter the Filer;
gRUNME,: G(et the workfile
   RUNME.TEXT
   and RUNME.CODE;
   (note that the comma acts as a
   carriage return <enter>)
q: Q(uit the Filer, and ...
r: R(un the program RUNME.CODE
```

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 after the program that contains it has finished executing. These routines are described in the following three sections.

Function REDIRECT

Function REDIRECT has the following declaration:

```
FUNCTION REDIRECT (EXEC_OPTIONS:
                   STRING):BOOLEAN;
```

EXEC_OPTIONS is an execution option string as defined above. 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.

Procedure EXCEPTION

Procedure EXCEPTION has the following declaration:

```
PROCEDURE EXCEPTION (STOPCHAINING:
                     BOOLEAN);
```

EXCEPTION turns off all 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).

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

Procedure CHAIN

Procedure CHAIN has the following declaration:

```
PROCEDURE CHAIN (EXEC_OPTIONS:STRING);
```

EXEC OPTIONS is an execution option string as defined above.

A call to CHAIN causes the System to eX(ecute EXEC_OPTIONS after the calling program (the "chaining program") has terminated. The effect is that 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 before control of the System is returned to the user.

A call to CHAIN with an empty string (for example, "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.

Individual Commands Alphabetically

Prompts for Filenames

Several of the p-System commands prompt for filenames. The conventions are the same for all responses to filename prompts throughout the p-System. A filename is typed in as letters, and followed by an *enter*. Before *enter* is typed, the name may be corrected by using *backspace* or *ctrl-backspace* 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 codefile 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? GRISWICH
Code file name? $
```

... 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? PRINTER:**

Responding to a filename prompt with just *enter* causes some default filename to be used (for example, *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).

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 native code. If there is no workfile, the user is prompted for a source file. The user is also prompted for a codefile and a listing file; the defaults for these are *SYSTEM.WRK.CODE and no listing file.

If the Assembler encounters a syntax error, it displays the error number, the source line in question, and (if the file *8086.ERRORS is present) an error message; finally, it displays the promptline:

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

The user has the choice of continuing assembly (*space*), aborting assembly (*esc*), or returning directly to the Editor to correct the source file (E).

The *Assembler Reference for the UCSD p-System* describes the assembler in detail.

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.

The *PASCAL Reference for the UCSD p-System* describes UCSD Pascal in detail. FORTRAN is described in the *FORTRAN-77 Reference for the UCSD p-System.*

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 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 3.

HALT

On the promptline:

H(alt.

Causes the p-System to stop execution. The only way to restart the System after a H(alt is by doing a hardware bootstrap.

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.

There is a SYSTEM.STARTUP on the STARTUP disk which accompanies the Beginners Guide for the UCSD p-System. You may also create your own SYSTEM.STARTUP. Some applications of this might be displaying reminders for the next session with the p-System, or crating 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 p-System to do an initialize. At initialize time, much of the p-System's internal data is rebuilt, and SYSTEM.MISCINFO is reread.

An I(nitialize will not clear any redirections (see Chapter 2), but any runtime error will.

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 5.

On the promptline:

M(on.

Redirecting the p-System's input (see Chapter 2) amounts to driving the p-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 p-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 p-System input to the monitor file with the I= execution option string.

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

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

The B(egin prompt starts a monitor. The user is prompted for a filename. This file becomes the monitor file. The user can then begin monitoring (and return to the System promptline) by using R(esume. If a monitor file has already been opened, an error message is displayed.

E(nd ends a monitor and saves the monitor file. The user may start another monitor with B(egin, or simply return to the System promptline with R(esume. If no monitor file is open, an error message is displayed.

A(bort ends the current monitor and does NOT save the monitor file. The user may start another monitor with B(egin, or return to the System promptline with R(esume.

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. If no monitor file is open, R(esume displays an error message to that effect (this is simply for the user's convenience).

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.

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

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.

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 Execution Option Strings in this Chapter). In the simplest case, this string contains nothing but the name of a codefile to be executed (as described in Chapter 2).

If the codefile cannot be found, the message Can't find *filename* 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 Chapter 2, 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.

NOTES

CHAPTER 3. FILES AND FILE HANDLING

Contents

Types of Files 3-3
File Formats 3-3
Volumes 3-5
Device numbers and descriptions 3-5
The Workfile 3-6
Filenames 3-6
Using the Filer 3-8
Prompts in the Filer 3-8
Names of Files 3-10
Filer Commands 3-12
Recovering Lost Files 3-42

NOTES

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:

FILES

.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.

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 (for example, 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 an *enter* (ASCII CR), and the first two may be a blank-compression pair. The optional blank-compression pair consists of an ASCII DLE followed by a byte whose value is 32+n, where n is the number of characters to indent. Text lines are typically 0..80 characters in length, so as to fit on the CRT.

Textfiles are considered to be unstructured files, and so the intrinsic SEEK will not work with them (SEEK is described in *PASCAL Reference for the UCSD p-System.*)

.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 p-System knows nothing about the internals of a datafile.

.FOTO files have an internal format which is of no concern to users. Foto files are manipulated by the TURTLEGRAPHICS Unit.

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 numbers and descriptions

Device Number	Volume ID	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTERM:	screen and keyboard without echo
4	<volume name>:	the System disk
5	<volume name>:	the alternate disk
6	PRINTER:	the parallel line printer
7	REMIN:	serial line input
8	REMOUT:	serial line output
9-12	<volume name>:	additional disk drives

The Workfile

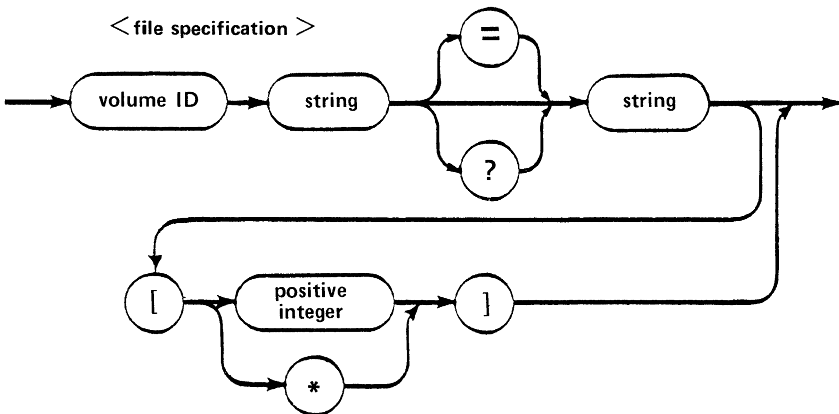
The workfile is described in Chapter 1. It is a scratchpad for creating files, and testing those files if they contain program text. The workfile is often stored temporarily in the files SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE. These may be either newly-created files, or copies of existing disk files which 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.

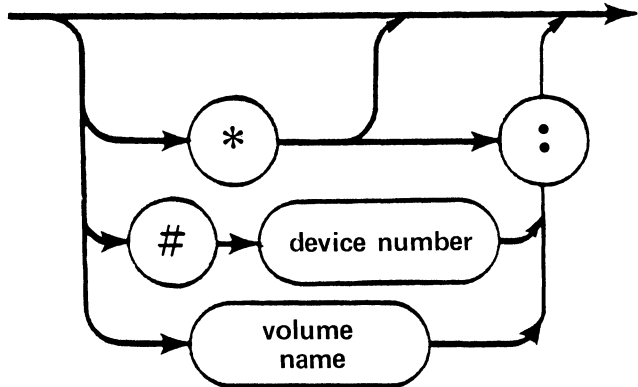
Filenames

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.



Volume ID syntax can be expanded as follows:



As shown in the table above, the volume name (for example, CONSOLE:) and the physical device number (for example, #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 3.1. The character * is shorthand for the volume ID of the System disk. The character : is shorthand for the volume ID of the default disk. The System disk and default disk are equivalent unless the default prefix has been changed. This can be done with the P(refix command (see below). The System disk is also called the root disk here and there. *#device number* is equivalent to the name of the volume in the drive at that time.

A legal filename can consist of up to 15 characters, including the .TEXT and .CODE suffixes, which are appended to a filename when the file is created, and reflect the internal organization of the file. Lower-case letters are translated to upper-case, and blanks and non-printing characters are removed

from the filename. Legal characters for filenames are the alphanumerics and the special characters -, /, , , and . . These special characters may be used as mnemonics to indicate relationships among files and/or to distinguish several related files of different types.

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

Using the Filer

Filer commands are described in detail below, in Chapter 3. They are listed 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.

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,? [C.11]
```

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

```
Filer: Q(uit, B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(ols? [C.11]
```

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 "Filename" previously described. Always follow file specifications with an *enter*. The wild cards are described 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 described, by separating the file specifications with commas, and terminating this file list with an *enter*. 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.

Names of Files

General Filename Syntax

For the Filer, filename syntax is the same as for the p-System in general, as described above in Chapter 3. 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.

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:

```
FILE1            6    1-Jan-82
GAME.TEXT        4    1-Jan-82
SYS.CODE        10    1-Jan-82
GAME.CODE        4    1-Jan-82
PROGRAM2.TEXT   12    1-Jan-82
G.FILE           5    1-Jan-82
```

Prompt: **Remove what file?**

Response: **Typing G= generates the message:**

```
MYDISK:GAME.TEXT      removed
MYDISK:GAME.CODE      removed
MYDISK:G.FILE         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.)

Example: Prompt: **Dir listing of what vol ?**

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

```
GAME.TEXT        4    1-Jan-82
PROGRAM2.TEXT   12    1-Jan-82
```

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.

FILES

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:

DISK2:RE.USE.TEXT --> SYSTEM:RE.USE.TEXT

Filer Commands

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

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 320 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.

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).

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 an *enter* 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,H.FILE**

... changes the name in the directory from F5.TEXT to H.FILE. Filetypes are originally determined by the filename; the C(hange command does not affect the filetype. In the above case, H.FILE 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, H.FILE would need to be renamed H.FILE.TEXT in order to be loaded into the workfile.

The user response #5:F5=,H.FILE=, 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 MYDISK: containing the files:

POEMS.TEXT
LETTER.TEXT
NAME.FILE
LIST.TEXT

Prompt: **Change what file?**

User response: **MYDISK:L=TEXT,S.L=BACK**

Causes the Filer to report:

MYDISK:LETTER.TEXT - -> S.LETTER.BACK
MYDISK:LIST.TEXT - -> S.LIST.BACK

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.

C(hange

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: **MYDISK:,WRKDISK:**

MYDISK: - -> WRKDISK:

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 1-Jan-82
New date?

The user may enter the correct date in the format given. After typing *enter*, the new date will be displayed. Typing only an enter 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 --83, the month by typing -Feb, etc. The entire month-name can be entered, but will be truncated by the Filer. The most common input is a single number, which is interpreted as a new day. For example, if the date shown is the first of January, and today is the second, the user would type 2 *enter*. 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.

E(xtended List

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

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

Since this command shows the complete layout of files and unused space on the disk, it is useful in conjunction with the M(ake command. Refer to "M(ake", and "Recovering Lost Files" 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:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
MYDISK:
F2.TEXT          28   1-Jan-82        6   512   Textfile
ABS12.CODE       18   1-Jan-82       34   512   Codefile
< UNUSED >       10                  52
ABS123            4   1-Jan-82       62   512   Datafile
HTYPE.CODE       21   1-Jan-82       66   512   Codefile
STATS.TEXT        8   1-Jan-82       87   512   Textfile
LET1.TEXT        18   1-Jan-82       95   512   Textfile
ASSEM.TEXT       20   1-Jan-82      113   512   Textfile
F.TEXT           24   1-Jan-82      133   512   Textfile
< UNUSED >       20                 157
STATS.CODE        6   1-Jan-82      177   512   Codefile
< UNUSED >      137                 183
9/9 files<listed/in-dir>, 153 blocks used, 167 unused, 137 in largest
```

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
ABS123.CODE
HYTYPER.CODE
STATIS.TEXT
LETTER1.TEXT
FILER.DOC.TEXT
STATIS.CODE

Prompt: **Get what file?**

Response: **STATIS**

The Filer responds with the message:

'Text & Code file loaded'

... since both text and code file exist. Had the user typed STATIS.TEXT or STATIS.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 ABS123, then that version would be loaded, regardless of whether text or code was requested. typing ABS123.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.

K(runch

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

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

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

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

K(runch

B(ad before doing a K(runch, unless all files are also backed up on a different disk.

If K(runch must move SYSTEM.PASCAL or SYSTEM.FILER on the boot disk, it will then display a prompt which asks you to reboot the p-System.

Example: Prompt: **Crunch what vol?**

Response: **MYDISK:**

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

Prompt: **From end of disk, block 320 ? (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.

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 valic volume ID for MYDISK (see Figure 5) in respons to the prompt,

Dir listing of what vol?

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11
MYDISK:
FILERDOC2.TEXT  38    1-Jan-82
ABS123.CODE     18    1-Jan-82
HYTPER.CODE     12    1-Jan-82
STATIS.TEXT      8    1-Jan-82
LETTER1.TEXT    18    1-Jan-82
FILERDOC1.TEXT  12    1-Jan-82
LETTER2.TEXT    12    1-Jan-82
STATIS.CODE      6    1-Jan-82
GAME.TEXT       10    1-Jan-82
TEMP             4    1-Jan-82
10/10 files<listed/in-dir>, 144 blocks used, 176 unused, 156 in larges
```

The bottom line of the display informs the user tha 10 files out of 10 files on the disk have been listed that 144 disk blocks have been used, that 176 disl blocks remain unused, and that the largest area available is 156 blocks.

The following is an example of the use of L(dir involving wildcards:

Prompt: **Dir listing of what vol ?**

User response: **#4:FIL=TEXT**

... generates the following display:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11
MYDISK:
FILERDOC2.TEXT      38     1-Jan-82
FILERDOC1.TEXT      12     1-Jan-82
2/10 files<listed/in-dir>, 56 blocks used, 176 unused, 156 in largest
```

The following L(dir example involves writing the directory subset to a device other than CONSOLE:

Prompt: **Dir listing of what vol ?**

User response: ***FIL=TEXT,PRINTER:<enter>**

causes:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
MYDISK:
FILERDOC2.TEXT      38     1-Jan-82
FILERDOC1.TEXT      12     1-Jan-82
2/10<listed/in-dir>, 56 blocks used, 176 unused, 156 in largest
```

... to be written to the printer.

The following L(dir example involves writing the directory subset to a block-structured device:

Prompt: **Dir listing of what vol ?**

User response: **#4:FIL=TEXT,#5:L.TEXT**

Creates the file L.TEXT on the volume associated with device 5. L.TEXT would contain:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
MYDISK:
FILERDOC2.TEXT      38     1-Jan-82
FILERDOC1.TEXT      12     1-Jan-82
2/10 files <listed/in-dir>, 56 blocks used, 176 unused, 156 in largest
```

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 "Recovering Lost Files").

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.

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 does create a new workfile.

A successful N(ew returns the message:

Workfile cleared

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., MYDISK:) in that device. If no volume is in the device when prefix is used, the default prefix remains the device number (e.g., #5:), and thereafter, any volume in the default device is the default volume.

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

Q(uit

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

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):

A.GAME.CODE
AUTO.TEXT
GAME.TEXT
AUTO.CODE

Prompt: **Remove what file?**

User Response: **AUTO.CODE**

... removes the file AUTO.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.

R(emove

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

Example: Prompt: **Remove what file?**

User Response: **A=CODE**

... causes the Filer to remove AUTO.CODE and A.GAME.CODE.

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

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

Saves the workfile under the filename specified by the user.

The entire file specificaiton 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

S(ave

assumes that the user wishes to save the workfile on another volume. For example, typing:

```
DISK2:PROG
```

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

```
MYDISK:SYSTEM.WRK.TEXT -->
                        DISK2:PROG.TEXT
```

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 *enter*. 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?**

User Response: **BACKUP:NAME.TEXT**

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.

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 "Recovering Lost Files" for a way to recover.

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

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

Example: Prompt: **Transfer what file?**

User Response: **FARKLE.TEXT**

Prompt: **To where?**

User Response: **PRINTER:**

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

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

The wildcard capability is allowed for T(ransfer. If the source file specification contains a wildcard character, and the destination file specification involves a block-structured device, then the destination file specification must also contain a

wildcard character. The subset-specifying strings in the source file specification will be replaced by the analogous strings in the destination file specification (henceforward known as replacement strings). Any of the subset-specifying or replacement strings may be empty. Remember that the Filer considers the file specification = to specify every file on the volume.

Example: Given the volume MYDISK containing the files I.1.TEXT, I.1A.TEXT, and I.2.TEXT, and the destination DISK2:

Prompt: **Transfer what file?**

User Response: **I=TEXT,DISK2:OLDI=BACK**

would cause the Filer to reply:

```
MYDISK:I.1.TEXT -->
                    DISK2:OLDI.1.BACK
MYDISK:I.1A.TEXT -->
                    DISK2:OLDI.1A.BACK
MYDISK:I.2.TEXT -->
                    DISK2:OLDI.2.BACK
```

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.

T(ransfer

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 ut so that it becomes an exact copy (including directory) of the source volume.

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

Prompt: **Transfer what file?**

User Response: **MYDISK:,EXTRA:**

Prompt: **Destroy EXTRA: ?**

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

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

V(olumes

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

A typical display might be:

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

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

W(hat

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

Example: **Workfile is DOC1:STUFF**

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).

eX(amine

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.

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 320 blks on the disk? (y/n)**

Response: N generates ...

Prompt: **# of blocks on the disk ?**
(this also appears if the disk was blank.)

Response: User types the number of blocks desired.

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

Recovering Lost Files

Sometimes a file is removed by accident, or its directory entry is written over for one reason or another. 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 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:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans D(ate,? [C.11]
WORK:
DATA             1   1-Jan-82         6   512   Datafile
< UNUSED >       1                    7
SYNTAX          14   1-Jan-82         8   512   Datafile
REM.CODE         4   1-Jan-82        22   512   Codefile
< UNUSED >      75                   26
MYFILE.TEXT     20   1-Jan-82       101   512   Textfile
< UNUSED >     199                  121
4/4 files<listed/in-dir>, 45 blocks used, 275 unused, 199 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:

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate,? [C.11]
WORK:
DATA             1   1-Jan-82         6   512   Datafile
< UNUSED >       1                    7
SYNTAX          14   1-Jan-82         8   512   Datafile
REM.CODE         4   1-Jan-82        22   512   Codefile
< UNUSED >      75                   26
MYFILE.TEXT     20   1-Jan-82       101   512   Textfile
MYFILE.CODE      4   1-Jan-82       121   512   Codefile
< UNUSED >     195                  125
5/5 files<listed/in-dir>, 49 blocks used, 271 unused, 195 in largest
```

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 Chapter 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 7. The following paragraph outlines the use of PATCH in this particular context; the reader should also refer to Chapter 7.

In order to look at a directory using PATCH, the user should eX(ecute PATCH, then type an *enter* 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 Chapter 7.

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 Chapter 7.

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 Chapter 7.

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

Example: L(ist directory will prompt:

Dir listing of what vol ?

Response: **MYDISK:, MYDISK: <enter>**
Response: **MYDISK:,: <enter>**

... either of these responses cause the first few blocks (approximately 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).

CHAPTER 4. THE SCREEN ORIENTED EDITOR

Contents

Introduction 4-3
The Concept of a Window into the File 4-3
The Cursor 4-4
The Promptline 4-4
Notation Conventions 4-5
The Editing Environment Options 4-5
Getting Started 4-5
Entering the Workfile and Getting a Program 4-6
Moving the Cursor 4-7
Using Insert 4-8
Using Delete 4-9
Leaving the Editor and Updating the Workfile 4-10
Using the Editor 4-11
Command Hierarchy 4-11
Repeat Factors 4-12
The Cursor 4-12
Direction 4-12
Moving the Cursor 4-13
Entering Strings in F(ind and R(eplace 4-15
Screen Oriented Editor Commands 4-16

EDITOR

NOTES

Introduction

This introduction describes the general environment of using the Screen Oriented Editor (called the Editor throughout this chapter). "Getting Started" is a tutorial for the novice, "Using the Editor" describes general conventions of the Editor, and "Screen Oriented Editor Commands" contains a detailed description of each command, with examples, in alphabetical order.

Ken Bowles' *Beginner's Guide for the UCSD p-System* is another good introduction to the Screen Oriented Editor, and we recommend that you use it.

The Concept of a Window into the File

The Screen Oriented Editor is specifically designed for use with the Video Display Terminal (or Cathode Ray Tube -- CRT). 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. The CRT which accompanies your IBM Personal Computer has 25 lines, and in the Editor one line is used for the prompts; thus, the Editor typically displays only 24 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.

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.

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(let F(ind I(nsrt J(mp K(ol R(plc Q(uit X(ch X(ap [E.7n]
```

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, *ent* means that the enter key should be typed at that point. Either lower or upper case may be used when typing Editor commands.

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 (S(et).

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 p-System running in front of you, and can duplicate the examples on your own.

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? (<ent> for no file <esc> to exit)

There are three ways to answer this question:

1) With the name, for example STRING1 *enter.* The file named STRING1.TEXT will now be retrieved. The file STRING1 could contain a program, also called STRING1, as in Figure 4-4-1. After typing the name, a copy of the text of the first part of the file appears on the screen.

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

Figure 4-1. Getting a program.

2) With an *enter*. 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.

Moving the Cursor

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

The cursor cannot be moved outside the text of the program. For example, after the N in BEGIN in Figure 4-2, 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.

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

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

Figure 4-2. Cursor movement (left-arrow).

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 4-3. The 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.*

```
aROGRAM STRING1
bEGIN
c WRITE(TOOWISE);
```

Figure 4-3. Cursor movement (down-arrow)

Now, using the *right-arrow*, move the cursor until it 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.

Using Insert

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

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

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

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

There remains one more important step. The choice at the end of the prompt line indicates that pushing the *etx* key (Ctrl-C) accepts the insertion, while

pushing the *esc* key rejects the insertion and the text remains as it was before typing I.

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

Figure 4-4. Screen after typing SMART.

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

Figure 4-5. Screen after Ctrl-C.

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

Figure 4-6. Screen after *esc*.

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

Using Delete

D(elete works like I(nsert. Having inserted SMART into the STRING1 program and having pushed *etx* (Ctrl-C), 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 CTRL-C 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.

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 4-7.

```
>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
```

Figure 4-7. Save a copy of the modified program.

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.

"Q(uit" explains in greater detail the options available at Q(uit.

Using the Editor

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.

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* command 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.

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.

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.

Moving the Cursor

The Cursor can be moved by a number of means. One obvious method is to use the four arrows on the key pad. Another method is to use traditional typewriter characters, i.e., *space bar, enter, tab,* and *backspace.* the former three are affected by the global direction. The arrow keys and *backspace* are not.

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

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

A variety of other commands reposition the cursor in addition to performing their specific actions. Thus, A(djust moves the cursor along with the entire line, C(opy and I(nsert move the cursor to the end of their insertions, and F(ind and R(eplace leave the cursor after their last successful hit. 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 usually every 8 spaces, starting at the left of the screen
enter	Moves cursor to the beginning of the next line

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

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

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, number or space 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.

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

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

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* (Ctrl-C).

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. Trailing blanks are not affected.

The margins can be altered with the S(et E(nvironment command. See "S(et E(nvironment".

The cursor is repositioned at the beginning of the last line adjusted. *etx* (Ctrl-C) is the only way to exit the A(djust command; *esc* will *not* work.

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.

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* (Ctrl-C); to escape, type *esc*.

In Figure 4-8:

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 *enter enter.* After the first enter the cursor moves to before the W in WRITELN, and WRITELN(TO BE.); disappears. After the second enter the cursor is before the W in WRITE, and that line has disappeared.

5) Now press *etx* (Ctrl-C). the program after deletion appears as shown in Figure 4-9.

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.

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

Figure 4-8. Before deletion.

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

Figure 4-9. After deletion.

F(ind

On the promptline: F(ind.

Repeat factors are allowed.

On entering Find, one of the promptlines in Figure 4-10 appears:

```
>Find[n]: L(it <target> =>
>Find[n]: T(ok <target> =>

{ Which line appears depends
on the global mode (see S(et) }
```

Figure 4-10. F(ind promptline.

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

F(ind finds the n-th occurrence of the *target* string, starting from the cursor's position and moving in the global direction (shown by the arrow at the beginning of the promptline). The cursor is positioned immediately after this occurrence. F(ind distinguishes between upper and lower case letters within the target string.

If you desire to search in other than the global mode (either Token or Literal), type the appropriate character (either L or T, respectively), before you enter the target string.

If the string is not present, the prompt:

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

... appears.

Example 1: In the STRING1 program (shown in Figure 4-11), 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.

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

Figure 4-11. Repeat F(ind.

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.

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 *enter*. 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* to remove one character at a time, or Ctrl-BACKSPACE 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.

Using Auto-indent

If Auto-indent is True, an *enter* causes the cursor to start the next line with an indentation equal to the indentation of the line above. If Auto-indent is False, an *enter* 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.

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 *enter*'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 *enter*'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 an *enter*. Important: This can only be done *immediately* after an *enter*.

Example 1: With Auto-indent true, the following sequence creates the indentation shown in Figure 4-12.

ONE *enter space space* TWO
enter THREE *enter backspace* FOUR

```
ONE
  TWO
  THREE
FOUR
```

Figure 4-12. Auto-indent true.

Example 2: With Filling True (and Auto-indent False) the following sequence creates the indentation shown in Figure 4-13:

```
ONCE UPON A TIME THERE- WERE.
```

(Very narrow margins have been used for simplicity.)

```
ONCE UPON A
TIME THERE-
WERE
```

Level of left margin

Figure 4-13. Filling true.

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

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 "M(argin". 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.

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.

K(olumn

On the promptline: K(ol.

Repeat factors are not allowed.

K(olumn displays the following prompt:

>**K(olumn:** <**vector keys**> <**etx**>

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). *etx* (Ctrl-C) 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.

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 4-14 has been M(argin'ed with the parameters on the left while the same paragraph in Figure 4-15 has been M(argin'ed with the parameters on the right.

Left-margin 0	Left-margin 10
Right-margin 40	Right-margin 40
Paragraph-margin 8	Paragraph-margin 0

```
        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 4-14. M(argin with left parameters.

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 4-15. M(argin with right parameters.

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.

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.

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.

Q(uit

On the promptline: Q(uit.

Repeat factors not allowed.

Q(uit displays the following prompt:

```
>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
```

Figure 4-16. Q(uit options.

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 Chapters 1, 2 and 3.

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:

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

Figure 4-17. W option.

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. If the file you are editing already existed before the edit session, you may specify $, which will write the file to the same name it had originally. Q(uit can be aborted at this point by typing *enter* instead of a filename; you will return to the Editor. If the file is written to disk, the Editor displays the following:

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

Figure 4-18. E option.

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.

R(eplace

On the promptline: R(plc.

Repeat factors are allowed.

On entering R(eplace one of the two promptlines in Figure 4-19 appears. In this example, a repeat factor of four is assumed:

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

Figure 4-19. R option.

R(eplace finds the target string (*targ*) exactly as F(ind would, and replaces it with the substitution string (*sub*). R(eplace distinguishes between upper and lower case letters in both the target string and the substitution string.

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.

R(eplace

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 4-20, 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 4-20 would change to those on the right-hand side.

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

Figure 4-20. R(eplace.

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>
```

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 an *enter*. 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:

```
Marker ovflw.
Which one to replace.
0)  name1
1)  name2
.   ...
.   ...
9)  name10
```

Figure 4-21. S(et M(arker.

... appears, it is necessary to eliminate one marker in order to replace it. Choose a number 0 through 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.

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:

```
>Environment: {options} <spacebar> to leave
    A(uto indent        True
    F(illing            False
    L(eft margin        1
    R(ight margin       80
    P(ara margin        6
    C(ommand ch         ^
    S(et tabstops
    T(oken def          True
    5415 bytes used, 26329 available.

    Patterns:
      <target> = 'one string', <subst> = 'another string'

Created March 1, 1982; Last updated March 1, 1982 (Revision 0).
Editor Version E.7n IV.02.
```

Figure 4-22. S(et E(nvironment Display

(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.

EDITOR

E(nvironment Options

A(uto indent:

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

F(illing:

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

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

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

C(ommand ch:

The Command character affects the M(argin command and the Filling option in I(nsert. Refer to those sections. Change the Command character by typing C followed by any character. For example, typing C* will change the Command character to *. This change will be reflected in the prompt. The Command Character was principally designed as a

convenience for users of text formatting programs whose commands are indicated by a special character at the beginning of a line.

S(et tabstops:

This command allows the positions of tab stops to be alerted. Tab stops default to every eight columns. When you enter this option, an 80 column dashed line will appear. Wherever a current tab stop exists, a letter (R, L, or D) will be displayed instead of a dash. By using the right or left arrow keys you may position the cursor at any position along the dashed line. (Alternatively, the C(ol command can be used to position the cursor at a specified column.) At any position, a tab stop may be created by typing R, L, or D (there is no difference between the different kinds of tab stops in the current release). A tab stop may be removed by positioning the cursor over it and typing N(o. ETX (Ctrl-C) is used to return to the E(nvironment level.

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 "Entering Strings in F(ind and R(eplace" for more information.

V(erify

Not on the promptline; type V for V(erify.

Repeat factors not allowed.

The current window is redisplayed.

eX(change

On the promptline; X(ch.

Repeat factors not allowed.

On entering eX(change the following promptline appears:

>eXchange: TEXT {<vector keys>} [<etx> <esc> CURRENT line]

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

For example, in the file in Figure 4-23, 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 4-24, with the cursor before the second S.

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

Figure 4-23. Before exchange.

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

Figure 4-24. After exchange.

etx (CTRL-C) 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, enter,* 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 INS key inserts one space at the cursor's location, and DEL deletes a single character at the cursor's location.

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.

CHAPTER 5. SEGMENTS, UNITS, LIBRARIES AND LINKING

Contents

Overview 5-5
Main Memory Management 5-5
Separate Compilation 5-6
General Tactics 5-8
Segments 5-11
Units 5-13
The Linker 5-18
Using the Linker 5-21
The Utility LIBRARY 5-23
Using LIBRARY 5-26
The Standard SYSTEM.LIBRARY Routines 5-26
The Screen Control Unit 5-27
PROCEDURE SC_Init 5-27
PROCEDURE SC_Clr_Cur_Line 5-27
PROCEDURE SC_Clr_Line (Y: integer) 5-27
PROCEDURE SC_Clr_Screen 5-27
PROCEDURE SC_Erase_to_EOL (X, Line: integer) 5-28
PROCEDURE SC_Eras_EOS (X, Line: integer) 5-28
PROCEDURE SC_Left 5-28
PROCEDURE SC_Right 5-28
PROCEDURE SC_Up 5-28
PROCEDURE SC_Down 5-28
PROCEDURE SC_Home 5-28
PROCEDURE SC_GOTO_XY (X, Line: integer) 5-28
FUNCTION SC_Find_X: integer 5-29
FUNCTION SC_Find_Y: integer 5-29

PROCEDURE SC_GetC_CH (VAR CH: char; Return_on_Match: SC_ChSet) 5-29
FUNCTION Space_Wait (Flush: Boolean): Boolean 5-29
FUNCTION SC_Prompt (Line: SC_Long_String; X_Cursor, Y_Cursor, X_Pos, Where: integer; Return_on_Match: SC_ChSet; No_Char_Back: Boolean; Break_Char: char): char 5-30
FUNCTION SC_Check_Char (VAR Buf: SC_Window; VAR Buf_Index, Bytes_Left: integer): Boolean 5-30
FUNCTION SC_Map_CRT_Command (VAR K_CH: char): SC_Key_Command 5-31
FUNCTION SC_Scrn_Has (What: SC_Scrn_Command): Boolean 5-31
FUNCTION SC_Has_Key (What: SC_Key_Command): Boolean 5-31
PROCEDURE SC_Use_Info (Do_What: SC_Choice; VAR T_Info: SC_Info_Type) 5-32
PROCEDURE SC_Use_Port (Do_What: SC_Choice; VAR T_Port: SC_TX_Port) 5-32
Unit COMMANDIO 5-33
Unit IBMSPECIAL 5-33
Function Button 5-35
Procedure Paddle 5-35
Procedure Note 5-35
Function Lightpen 5-36
Procedure Setkeys (tableptr: key_ptr) ... 5-36
Procedure Videomode (mode: threebits) 5-37
Procedure Setfont (table: font_ptr) 5-38
Procedure Bkgnd_Color (color: fourbits) 5-38

Procedure Palette (color: onebit) 5-39
Procedure Settime
(hour, minute: integer) 5-39
Procedure Gettime 5-39
The Turtle Graphics Unit 5-40
Procedure Move (distance: real) 5-42
Procedure Moveto (x,y: real) 5-42
Procedure Turn (rotation: real) 5-42
Procedure Turnto (heading: real) 5-43
Proedure Pen_color (shade: integer) ... 5-43
Procedure Pen_mode
(mode: integer) 5-44
Function Turtle_x: real 5-45
Function Turtle_y: real 5-45
Function Turtle_angle: real 5-45
Procedure Activate_Turtle 5-45
Procedure Fillscreen 5-46
Procedure Background 5-47
Procedure Align_cursor (x,y: real) 5-47
Procedure Display_scale 5-48
Function Aspect_ratio: real 5-49
Function Create_figure 5-50
Procedure Delete_figure 5-51
Procedure Getfigure 5-51
Procedure Putfigure 5-52
Procedure Viewport 5-53
Pixels 5-53
Function Read_pixel 5-54
Procedure Set_pixel 5-54
Function Read_figure_file 5-55
Function Write_figure_file 5-55
Function Load_figure 5-55
Function Store_figure 5-55

NOTES

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.

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.

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 the 8086/88/87 Interpreter) 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 (for example, 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 the *Assembler Reference for the UCSD p-System.*

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 the *PASCAL Reference for the UCSD p-System*, and inform the Operating System by including the codefile's name in a "library text file." The default library text file is *USERLIB.TEXT, but this default can be changed by an execution option. See "Units" and Chapter 2.

For the use of units in FORTRAN, refer to the *FORTRAN-77 Reference for the UCSD p-System.*

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 be 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.

"Segments" and "Units" of this chapter describe the syntax of using UNITs and SEGMENT routines in Pascal. For information on other languages, refer to the appropriate manual.

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.

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 (I: 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}.
```

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 the *PASCAL Reference for the UCSD p-System*).

```
UNIT <unit identifier>;

INTERFACE
     USES <unit identifier list>;
     <constant definitions>;
     <type definitions>;
     <variable declarations>;
     <routine headings>;

IMPLEMENTATION
     USES <unit identifier list>;
     <label declarations>;
     <constant definitions>;
     <type definitions>;
     <variable declarations>;
     <routine declarations>;

[ BEGIN
     <initialization statements>
     ***;
     <termination statements>]
END
```

The INTERFACE part may only contain routine headings -- no bodies. The bodies of routines declared in the INTERFACE part are fully defined in the IMPLEMENTATION part, much as FORWARD procedures are fully defined apart from their original declaration.

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

An INTERFACE part may not contain $Include files (see the *PASCAL Reference for the UCSD p-System* for information on Include files). An INTERFACE part may be contained within an $Include file, provided that all of the INTERFACE is in the $Include file; i.e., an INTERFACE part may not cross an $Include file boundary. Note that IMPLEMENTATION terminates an INTERFACE part, so that if an INTERFACE part is contained in a $Include file, the $Include file must contain both the reserved words INTERFACE and IMPLEMENTATION.

Example:

```
UNIT GOLE1;
INTERFACE
   {$I INTER DECS}
IMPLEMENTATION
   ...
END;
```

```
UNIT GOLE2;
{$I INTER_PART}
IMPLEMENTATION
...
END;
```

... are not legal forms of a UNIT, while the following outline is:

```
UNIT GOLE3;
{$I WHOLE UNIT}
;
```

The *initialization statements* and *termination statements* are optional sections of code. Initialization statements, if present, are executed before any of the code in a host that USES the UNIT is executed,

and termination statements, if present, are executed after the host's code has terminated.

Initialization statements are separated from termination statements by the line ***;. Either the section of initialization statements, or the section of termination statements, or both, may be empty.

Example: The following are all legal code bodies of a UNIT:

```
END {there is no initialization or
     termination code};

BEGIN
     {this is initialization code}
     INIT_ARRAYS;
     FLAG := FALSE;
     COUNT := 23;
     ***;
     {this is termination code}
     SEMINIT ( LIGHT, 0 );
END {UNIT};

BEGIN
     ***;
     {this is all termination code}
     INIT_ARRAYS;
     FLAG := FALSE;
     COUNT := 23;
     SEMINIT ( LIGHT, 0 )
END {UNIT};

BEGIN
     {this is all initialization code}
     INIT_ARRAYS;
     FLAG := FALSE;
     COUNT := 23;
     SEMINIT ( LIGHT, 0 )
END {UNIT};
```

The statement part of a UNIT should not contain GOTO statements which branch around the ***; separator: the effect of executing such statements is not fully predictable.

A UNIT's statement part may contain statements of the form: EXIT(PROGRAM) (EXIT(*unitname*) is not allowed). An EXIT(PROGRAM) in the initialization code has the effect of skipping the remainder of the initialization code (if any) and the host's code: execution proceeds with the UNIT's termination section. An EXIT(PROGRAM) in the termination code skips the remainder of the termination code (there may be termination code from other hosts still waiting to execute -- the EXIT does not abort the execution of these other termination sections).

To use one or more UNITs, a program must name them in a USES declaration immediately following the program heading (before the *block*). Upon encountering a USES declaration, the compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all identifiers declared in the INTERFACE part are global. Name conflicts may arise if the host defines an identifier already defined in the UNIT.

A UNIT may also USE another UNIT. In this case, the USES declaration may appear at the beginning of either the INTERFACE part or the IMPLEMENTATION part. Since USES may be nested, if they appear in the INTERFACE part, the ordering of a USES declaration may be important: if UNIT_A USES UNIT_B, then the host must specify that it USES UNIT_B before it USES UNIT_A.

Routines declared in the INTERFACE part must not be SEGMENT routines, but SEGMENT routines can be declared in the IMPLEMENTATION part. (Declaring SEGMENTs within UNITs is subject to the same ordering as within a main program; see "Segments" in this chapter).

For purposes of listing a program, the Compiler treats an INTERFACE section as an include level. Thus, $Include file nesting is restricted within the scope of a USES declaration.

The UCSD p-System will compile a Pascal program, a single UNIT, or a string of UNITs (separated by semicolons). A Pascal program may define a UNIT in-line. An in-line UNIT definition must appear between the program heading and the *block*. This has the advantage of simplicity, but if changes are made to either the program or the UNIT, both must be recompiled.

UNITs need not be explicitly linked together. At compile-time a USEd UNIT's INTERFACE part must be referenced by the Compiler. If the UNIT's source is in the host program's source, or if the UNIT's code is in *SYSTEM.LIBRARY, nothing more need be specified. If the UNIT's code resides in a different file (a "user library"), the $U Compiler directive must be used to specify which file (see the *PASCAL Reference for the UCSD p-System*).

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 Chapter 2).

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 in 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.

The use of UNIT-style mechanisms in the p-System's other high-level languages is discussed in the documentation for each particular language. External linkages involving assembled routines are discussed in the *Assembler Reference for the UCSD p-System* and in the next section.

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 into a Pascal or FORTRAN program. EXTERNAL routines are routines (procedures, functions, or processes) that are written in an assembly language and conform to the p-System's calling and parameter-passing protocols. They are declared EXTERNAL in the

host program, and must be linked before the program is run. The Linker may also be used to link together separately assembled pieces of a single assembly program.

The Linker is a program of the sort called a "link editor". It stitches code together by installing the internal linkages that allow various pieces to function as a unified whole.

When a program which must be linked is R(un, the Linker will automatically search *SYSTEM.LIBRARY for the necessary external routines. In all other cases (i.e., the user used eX(ecute instead of R(un or the library is not SYSTEM.LIBRARY), the user is responsible for "manually" linking the code before executing it.

When the Linker is called automatically and cannot find the needed code in *SYSTEM.LIBRARY, it will respond with an error message:

Proc,
Func,
Global,
or Public <identifier> undefined

To link code "by hand", call the Linker by typing L at the command level.

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 **enter* or any filename that ends in a .). The response **enter* or simply *enter* 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 *enter* is typed. Responding **enter* opens *SYSTEM.LIBRARY. The success of opening each library file is reported.

Example (underlined portions are user input):

```
Lib file? *<enter>
Opening *SYSTEM.LIBRARY
Lib file? FIX.8<enter>
No file FIX.8.CODE
Type <sp>(continue), <esc>(terminate)
Lib file? FIX.9<enter>
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 *enter*, 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 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.

The Utility LIBRARY

LIBRARY.CODE is a utility program that allows the user to group separate compilations (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, 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.

Using LIBRARY

When LIBRARY is executed, a prompt asks for an output filename. The filename must end in .CODE if the output file is to be an executable codefile. 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 if necessary.

Example: The user specifies SCREENOPS.CODE as an input file. LIBRARY displays the following:

```
Library: N(ew, 0-9(slot-to-slot, E(very,
              S(elect, C(omp-unit

Input file? SCREENOPS<enter>
   0  u SCREENOP    921                     8
   1  s SEGSCINI    416                     9
   2                                       10
   3                                       11
   4                                       12
   5                                       13
   6                                       14
   7                                       15

Write to what file? NEW.CODE<enter>
   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 *enter*). Simply typing *enter* exits LIBRARY without writing a copyright notice.

T(og toggles a switch which determines whether or not INTERFACE parts of UNITs are copies 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 *enter* 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 already filled, then LIBRARY searches for the next available slot and places the code there. If, for any code segment, an identical code segment already exists in the output file, that segment is not copied over.

S(elect causes LIBRARY to prompt the user for which code segments to transfer. For each code segment not already in the output file, LIBRARY prompts: Copy from slot #_?. A Y or N causes the segment to be copied or passed by, an E causes the remainder of the code segments to be transferred (as in E(very), a *space* or *enter* 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.

The Standard SYSTEM.LIBRARY Routines

The SYSTEM.LIBRARY file, as it is delivered to users of the IBM Personal Computer, contains three units. These are unit LONGOPS, the IBMSPECIAL, and the TURTLEGRAPHICS unit. The EXTRAS diskette contains some additional units including COMMANDIO.CODE and SCREENOPS.CODE (which are of interest here).

Unit LONGOPS is used automatically when Long Integers are declared within a program. (See the *PASCAL Reference for the UCSD p-System* for information about Long Integers.) The next four sections describe the routines available to users within the other four units.

The Screen Control Unit

The Screen Control Unit is located on the EXTRAS diskette and is called SCREENOPS.CODE. This section describes how the Screen Control Unit may be used to perform various CRT-related tasks.

In order to use the Screen Control Unit, a program must contain the proper USES declaration:

```
USES {$U screenops.code} SCREENOPS;
```

or

```
USES {$U #5:screenops.code} SCREENOPS;
```

SCREENOPS is actually an Operating System unit, but the copy of SCREENOPS within SYSTEM.PASCAL does not contain the Interface section (this saves disk space). During compilation the copy of SCREENOPS within SCREENOPS.CODE must be available, but during

runtime, just having SYSTEM.PASCAL on the disk would be sufficient. If desired, SCREENOPS may be added to SYSTEM.LIBRARY using the LIBRARY utility. In this case the {$U screenops.code} should be omitted from the above declaration.

All of the routines described in this section may be called from your program. The text ports mentioned below are rectangular portions of the screen which may be defined to be of a different size than the real screen. Where text ports are mentioned in this section, the entire screen should be understood to be the default.

PROCEDURE SC_Init;

Usually this procedure is only called by the Operating System. It initializes all the Screen Control tables and variables.

PROCEDURE SC_Clr_Cur_Line;

Erases the current line.

PROCEDURE SC_Clr_Line (Y: integer);

Clears line number Y within the current text port.

PROCEDURE SC_Clr_Screen;

Clears the screen.

PROCEDURE SC_Erase_to_EOL (X, Line: integer);

Starting at position (X, Line) within the current text port, everything to the end of the line is erased.

PROCEDURE SC_Eras_EOS (X, Line: integer);

Starting at position (X, Line) within the current text port, everything to the end of the screen is erased.

PROCEDURE SC_Left;

Moves the cursor one character to the left.

PROCEDURE SC_Right;

Moves the cursor one character to the right.

PROCEDURE SC_Up;

Moves the cursor one line up (in the same column).

PROCEDURE SC_Down;

Moves the cursor one line down.

PROCEDURE SC_Home;

Moves the cursor to position 0,0 within the current text port.

PROCEDURE SC_GOTO_XY (X, Line: integer);

Moves the cursor to position (X, Line).

FUNCTION SC_Find_X: integer;

Returns the column position of the cursor, relative to the current text port.

FUNCTION SC_Find_Y: integer;

Returns the row position of the cursor, relative to the current text port.

PROCEDURE SC_GetC_CH (VAR CH: char; Return_on_Match: SC_ChSet);

SC_ChSet is a SET OF CHAR. This procedure repeatedly reads from the keyboard into CH until CH is equal to a member of Return_on_Match. The characters that you pass in this set should all be capitals (if they are alphabetic). If a lower case alphabetic character is received from the keyboard, it will be translated into upper case before it is compared to the characters within Return_on_Match.

FUNCTION Space_Wait (Flush: Boolean): Boolean;

This function repeatedly reads from the keyboard until a *space* or the ALTMODE character is received. Before doing this it does a UNITCLEAR(1) if Flush is TRUE, and writes Type *space* to continue. It returns TRUE if a *space* was not read.

FUNCTION SC_Prompt (Line: SC_Long_String; X_Cursor, Y_Cursor, X_Pos, Where: integer; Return_on_Match: SC_ChSet; No_Char_Back: Boolean; Break_Char: char): char;

This function displays the promptline, Line (SC_Long_String is a STRING [255]) in the current text port at (X_Pos, Where). The cursor is placed at (X_Cursor, Y_Cursor) after the prompt is printed. If X_Cursor is less than 0, the cursor is placed at the end of the prompt. If the prompt is too large to fit within the current text port, it is broken up into several pieces, but only at the Break_Char -- the user can view different parts of the prompt (cycling through them) by typing '?'. If a character is being prompted for, No_Char_Back should be sent as false. The keyboard is repeatedly read until the character read matches one within Return_on_Match.

FUNCTION SC_Check_Char (VAR Buf: SC_Window; VAR Buf_Index, Bytes_Left: integer): Boolean;

While a string is being read, this function may be called to see if a *backspace* or a DEL has been read. If so, the input buffer is altered accordingly, and TRUE is returned. Buf is a line on the screen, Buf_Index indicates the cursor position within Buf, and Bytes_Left is the number of characters to the right of the cursor.

FUNCTION SC_Map_CRT_Command (VAR K_CH: char): SC_Key_Command;

SC_Key Command is a type consisting of the following elements: (SC_Backspace_Key, SC_DC1_Key, SC_EOF_Key, SC_ETX_Key, SC_Escape_Key, SC_DEL_Key, SC_Up_Key, SC_Down_Key, SC_Left_Key, SC_Right_Key, SC_Not_Legal). The character passed is mapped into one of these elements.

FUNCTION SC_Scrn_Has (What: SC_Scrn_Command): Boolean;

SC_Scrn_Command is a type consisting of the following elements: (SC_Home, SC_Eras_S, SC_Eras_EOL, SC_Clear_Lne, SC_Clear_Scn, SC_Up_Cursor, SC_Down_Cursor, SC_Left_Cursor, SC_Right_Cursor). This function returns TRUE if the CRT has the control character passed.

FUNCTION SC_Has_Key (What: SC_Key_Command): Boolean;

SC_Key_Command consists of the elements listed in the description of SC_Map_CRT Command above. This function returns true if the CRT generates the keyboard character passed.

PROCEDURE SC_Use_Info (Do_What: SC_Choice; VAR T_Info: SC_Info_Type);

This function is used to pass information back and forth between a program and the Screen Control Unit. Do_What may either be SC_Get or SC_Give, and indicates whether the program is getting or giving information to the Screen Control Unit. T_Info contains various items to be either passed or received. The following information is contained within T_Info:

```
SC_Version: string;
SC_Date: PACKED RECORD
        Month: 0..12;
        Day: 0..31;
        Year: 0..99;
     END;
Spec_Char: SET OF char; (*Chars not to echo*)
Misc_Info: PACKED RECORD
        Height, Width: 0..255;

Can_Break,Slow,XY_CRT,LC_CRT,
        Can_UpScroll, Can_DownScroll
          : Boolean;
      END;
```

PROCEDURE SC_Use_Port (Do_What: SC_Choice; VAR T_Port: SC_TX_Port);

This function works like SC_Use_Info above. The contents of T_Port are either passed or received from the Screen Control Unit. T_Port contains the following information:

```
Row, Col,
Height, Width,
Cur_X, Cur_Y : integer;
```

Unit COMMANDIO

COMMANDIO is located on the EXTRAS diskette and is called COMMANDIO.CODE. COMMANDIO is a unit within the Operating System. But, like SCREENOPS, the copy of COMMANDIO within SYSTEM.PASCAL does not have the Interface section (again to save disk space). The copy of COMMANDIO within COMMANDIO.CODE does have the Interface section and can be used by a program with the proper declaration:

```
  USES {$U commandio.code} COMMANDIO;
or
  USES {$U #5:commandio.code} COMMANDIO;
```

The using program may then call the routines REDIRECT (which is used to redirect output and input), EXCEPTION (which turns off redirection), and CHAIN (which allows several programs to be chained together). These routines are described in Chapter 2.

Unit IBMSPECIAL

Unit IBMSPECIAL is located within SYSTEM.LIBRARY and contains routines that are useful for various tasks including writing game programs. In order to use this unit, your program should contain the declaration:

```
USES IBMSPECIAL;
```

The standard SYSTEM.LIBRARY should be on the boot disk at compilation time as well as at run time. The following is the Interface section from this unit:

```
Unit IBMspecial;
Interface
Type
     pitch_range=0..12;
     octave_range=0..7;
     onebit=0..1;
     twomax=0..2;
     twobits=0..3;
     threebits=0..7;
     fourbits=0..15;
     font_pattern=packed array[0..63] of
       boolean;
     font_table=array[128..255] of
       font_pattern;
     font_ptr=^font_table;
     str_ptr=^string;
     key_table=array[1..16] of str_ptr;
     key_ptr=^key_table;

Function Button (select:twobits): Boolean;
Procedure Paddle (select:twobits;
                              Var result:Integer);
Procedure Note (pitch:pitch_range;
                              octave:octave_range;
                              duration:integer);
Function Lightpen (Var charxpos, charypos,
                              pixelxpos, pixelypos:
                              integer): Boolean;
Procedure Setkeys (tableptr:key_ptr);
Procedure Videomode (mode:threebits);
Procedure Setfont (table:font_ptr);
Procedure Bkgnd_Color (color:fourbits);
Procedure Palette (color:onebit);
Procedure Settime (hour, minute: Integer);
Procedure Gettime (Var hour, minute:
                              Integer);
Procedure Select_Printer (unitnum:twomax);
Procedure Select_Remote (unitnum:onebit);
```

Function Button (select: twobits): Boolean;

Returns true if the selected game button (0..3) is depressed. These buttons are located on the paddle controls which plug into the IBM Personal Computer Game Control Adapter.

Procedure Paddle (select: twobits; VAR result: integer);

Returns the position of the paddle control mechanism specified by the parameter Select. The value of the parameter Result contains this position.

Procedure Note (pitch: pitch_range; octave: octave_range; duration: integer);

Causes the speaker to sound a tone. These are the meanings of the parameters passed to this procedure:

pitch:
- 0 = rest
- 1 = C
- 2 = C#
- 3 = D
- 4 = D#
- 5 = E
- 6 = F
- 7 = F#
- 8 = G
- 9 = G#
- 10 = A
- 11 = A#
- 12 = B

octave:
- 0 = lowest
- 7 = highest

duration: milliseconds, taken as an unsigned integer.

SEG/UNITS/LINK

Function Lightpen (VAR charxpos, charypos, pixelxpos,pixelypos: integer): Boolean;

Returns true if the lightpen is activated. The parameters return the position of the lightpen:

charxpos is the X position, 0..79
charypos is the Y position, 0..24
pixelpos is the pixel X pos., 0..639
pixelypos is the pixel Y pos., 0..199

Procedure Setkeys (tableptr: key_ptr);

Allows the programmer to define the 16 function keys on the Personal Computer keyboard. There are 10 such keys on the left-hand side of the keyboard (labeled F1 - F10), and an additional six keys on the number pad:

11 Home
12 PageUp
13 End
14 PageDown
15 INS
16 DEL

These keys return default codes (as described in Appendix E) unless they are redefined by a call to SetKeys. SetKeys must be passed a pointer to a table. This table is itself an array (1..16) of pointers to strings. Each entry in the table corresponds to one of the 16 function keys: when the key is pressed, the string that the table entry points to is written, just as if all the characters it contains were typed by hand. A table entry should not be NIL if the corresponding key is to be used.

When the program terminates, the definitions created by SetKeys disappear.

Procedure Videomode (mode: threebits);

Sets the mode of the screen. The screen may be set to 80 or 40 columns across. And it may be set to alpha-numeric mode or graphic mode. It is necessary for the screen to be in graphic mode if graphics are to be used. Standard alpha-numeric characters can be sent to the screen when it is in graphics mode, but this is not recommended since more time is required. Parameter Mode indicates the following modes:

0: 40 x 25 b & x alpha-numeric mode
1: 40 x 25 color alpha-numeric mode
2: 80 x 25 b & w alpha-numeric mode
3: 80 x 25 color alpha-numeric mode
4: 40 x 25 color graphic mode (320 x 200)
5: 40 x 25 b & w graphic mode (320 x 200)
6: 80 x 25 b & w graphic mode (640 x 200)
7: Black/White Card

The cursor is not visible in modes 4, 5 and 6.

Note: Mode 7 is for those terminals set up to display only in black and white. They are always in mode 7. All other configurations (i.e. those which may display in color) can never be in mode 7.

Procedure Setfont (table: font_ptr);

The "upper" 128 ASCII characters (80..FF) may be redefined by the user with a call to Setfont. Setfont is passed a pointer to a table. The table contains 128 entries (128..255), each of which is an array of bits (0..63). This Boolean array defines a character: the bits are arranged in the following pattern:

```
 7   ...   0
15   ...   8
23   ...  16
31   ...  24
39   ...  32
47   ...  40
55   ...  48
63   ...  56
```

... where each entry represents a single pixel. This use of the font applies to alphanumeric mode, NOT graphics mode. The definitions disappear when the program that called Setfont terminates.

Procedure Bkgnd_Color (color: fourbits);

Sets the background color in the graphic mode. It sets the border color in the alpha-numeric mode. Parameter Color indicates the following colors:

```
1  =  Blue
2  =  Green
4  =  Red
8  =  High Intensity
```

These numbers may be added together to produce combinations.

Procedure Palette (color: onebit);

Sets the colors to be used in the graphics mode. If Parameter Color is 0, the following are the colors that will be produced:

0 = Background Color
1 = Green
2 = Red
3 = Yellow

If Parameter Color is 1, the following are the colors that will be produced:

0 = Background Color
1 = Cyan
2 = Magenta
3 = White

These are the colors that will be indicated by a variable of Type Color within the Turtlegraphics Unit (see "The Turtle Graphics Unit" in this chapter).

Procedure Settime (hour, minute: integer);

Sets the clock to correspond to the indicated hour and minute.

Procedure Gettime (VAR hour, minute: integer);

Gets the time of day. This procedure will return meaningless information if Settime has not been called first. After rebooting, Settime must be called again if Gettime is to return valid information.

The Turtlegraphics Unit

Turtlegraphics is a package of routines for creating and manipulating images on the graphic display. These routines can be used to control the background of the screen, draw figures, alter old figures, and display figures using viewports and scaling. It also contains routines that allow the user to save figures in disk files and retrieve them.

The simplest Turtlegraphics routines are intentionally very easy to learn and use. Once the user is familiar with these, more complicated features (such as scaling and pixel addressing) should present no problem.

A "pixel," by the way, is a single "picture element" or point on the display.

Turtlegraphics allows the user to create a number of "figures," or drawing areas. One such figure is the display screen itself, and other figures may be saved in memory. Each figure has a turtle of its own. The size of a figure may be set by the user (it does not need to be the same size as the actual display). See "Figures and the Port" in this chapter.

The actual display is addressed in terms of a display scale, which may be set by the programmer. This allows the user's own coordinates to be mapped into pixels on the display. All other figures are scaled by the global display scale. See "Scaling" in this chapter.

The programmer may define a "viewport," or window on the display. This limits all graphic activity to within that port. See "Figures and the Port".

Each subsection below is divided into two parts. The first part is an overview of the topic at hand, and the second part consists of descriptions of the relevant Turtlegraphics routines.

For quick reference, "Routine Parameters" contains a listing of the Interface part of the Turtlegraphics unit.

"Sample Program" contains a sample program that illustrates a number of the Turtlegraphics routines.

The Turtlegraphics unit uses the IBMSPECIAL unit described in "Unit IBMSPECIAL" in this chapter.

The Turtle

The "turtle" is an imaginary creature that resides on the display screen. It carries with it a "pen," and can be made to draw lines by moving it about the display. The possible movements of a turtle are:

> move in a straight line (Move);
>
> move to a particular point on the display (Moveto);
>
> turn, relative to the current direction (Turn);
>
> turn to a particular direction (Turnto).

Thus, the turtle draws straight lines in some given direction. The color of the lines it draws can be specified (Pen_color), and so can the nature of the line drawn (Pen_mode).

Wherever the turtle is located, its position and direction can be ascertained by three functions: Turtle_x, Turtle_y, and Turtle_angle.

Note that the turtle may be moved *anywhere:* it is not limited by the size of the figure or the size of the display. But only movements within the figure will be visible.

To use the turtle in a figure other than the actual display, the programmer may call Activate_Turtle. We will discuss new figures in "Figures and the Port" in this chapter.

The remainder of this section describes the routines that handle the turtle. Since this section is meant to double as a reference, some of the routine descriptions mention features we have not yet discussed. Just skim over anything you do not yet understand.

Procedure Move (distance: real);

Moves the active turtle the specified distance along its current direction. The turtle leaves a tracing of its path (unless the drawing mode is 'nop'). The distance is specified in the units of the current display scale (see "Scaling"). The movement will be visible unless the current turtle is in a figure that is not currently on the display.

Procedure Moveto (x,y: real);

Moves the active turtle in a straight line from its current position to the specified location. The turtle leaves a tracing of its path (unless the drawing mode is nop). The x,y coordinates are specified in the units of the current display scale.

Procedure Turn (rotation: real);

Turns the active turtle by the amount specified (in degrees). A positive angle turns the turtle counterclockwise, and a negative angle turns it clockwise.

Procedure Turnto (heading: real);

Sets the direction (the "heading") of the active turtle to a specified angle. The angle is given in degrees; zero (0) degrees faces the right-hand side of the screen, and ninety (90) degrees faces the top of the screen.

Procedure Pen_color (shade: integer);

Selects the color with which the active turtle traces its movements (unless the pen mode is nop). This color remains the same until Pen_color is called again.

The color of the pen depends on the way the video display is set. See "Unit IBMSPECIAL". If the palette is set to 0, the colors are:

0 = Wildcard Color
1 = Green
2 = Red
3 = Yellow

If the palette is set to 1, the colors are:

0 = Wildcard Color
1 = Cyan
2 = Magenta
3 = White

If the display is in the black and white graphics mode 5, the color indicates:

0 = Wildcard Color
1 = Gray 1
2 = Gray 2
3 = Gray 3

If the display is in the black and white graphics mode 6, the color indicates:

0 = Black
1 = White

We use the term "wildcard" to refer to the background color of the display that is set by the procedure Bckgnd_Color in IBMSPECIAL. This is a function of the display hardware, and might be called a "hard" background. In Turtlegraphics, each individual figure may have its own "soft" background color, which we refer to simply as the "background color" (as in the discussion below). See "The Display".

Procedure Pen_mode (mode: integer);

Sets the active turtle's drawing mode. This mode does not change until Pen_mode is called again.

These are the possible modes:

0 = Nop - does not alter the figure.

1 = Substitute - writes the current pen color.

2 = Overwrite - writes the current pen color.

3 = Underwrite - writes the current pen color. When the pen crosses a pixel that is not of the background color, that figure is not overwritten.

4 = Complement - the pen complements the color of each pixel that it crosses. (The complement of a color is its opposite: the complement of the complement of a color is the original color.)

Values greater than 4 are treated as Nop.

(These descriptions apply to movements of the turtle. They have a more complex meaning when a figure is copied onto a figure that is already displayed: see "Figures and the Port".

Function Turtle_x : real;

Returns a real value that is the x-coordinate of the active turtle, in units of the current Display_scale.

Function Turtle_y : real;

Returns a real value that is the y-coordinate of the active turtle, in units of the current Display_scale.

Function Turtle_angle : real;

Returns a real value that is the direction (in degrees) of the active turtle.

Procedure Activate_Turtle (screen: integer);

Specifies to which figure subsequent Turtlegraphics commands are directed. Each invocation of this procedure puts the previously active turtle to sleep and awakens the turtle in the designated figure. When Turtlegraphics is initialized, the turtle in the actual display is awake.

The Display

The color of the display itself (or any other figure) depends in part on settings determined by routines within the IBMSPECIAL unit. The size, background color, and color range of the display are set by Videomode, Bckgnd_Color, and Palette, and these settings apply whenever Turtlegraphics is used.

We refer to the initial background of the display as the wildcard color. The wildcard color (color 0) is set by Bckgnd_Color in IBMSPECIAL. The default is black. The background color of a Turtlegraphics figure may be changed by the programmer with a call to Background. This "soft" background applies when drawing mode is used, as indicated above.

A figure can be filled with a single color (not necessarily the background color) by calling Fillscreen.

> **Note:** When Turtlegraphics is initialized, the video mode is set to 4. The programmer may call Videomode in IBMSPECIAL to change this, but the display is cleared, and if the new mode is other than 4, Display_scale must be called immediately to re-initialize the display. There is no termination code in Turtlegraphics to reset the video mode to whatever it was before the user program started (since it is not possible to determine what that mode was). It is therefore the responsibility of the programmer to reset the video mode, if desired, at the end of his or her program with another call to Videomode.

Procedure Fillscreen (screen: integer, shade: integer);

Fills the specified figure ("screen") with the specified color ("shade"). If screen = 0, which indicates the actual display screen, then only the current viewport is shaded. For user-created figures, the entire figure is shaded.

Procedure Background (screen: integer; shade: integer);

Specifies the background color for a figure. The initial background color of all figures is the wildcard color.

Labels

It is possible to draw legends, labels, and so forth on the display while using the Turtlegraphics unit. Position the cursor in the desired location with the Align_cursor routine, and write to the console using WRITE or WRITELN.

Procedure Align_cursor (x,y: real);

Positions the cursor at the specified location. The x,y co-ordinates are specified in the units of the current display scale (see "Scaling"). The cursor is positioned over the x,y point. Text may now be written to the screen in the usual manner. It is not confined to the viewport. However, if x,y falls outside the current viewport, the cursor is initially positioned at the nearest point within the viewport, and writing begins at that location.

Scaling

When a programmer wishes to display data without altering the input data itself, it is possible to set scaling factors that translate data into locations on the display. This is done with Display_scale. The display scale applies globally to all figures.

Because of the shape of the actual display, data for particular shapes (especially curved figures) might become distorted when using a "straight" display scale. In this case, the function Aspect_ratio can be used to preserve the "squareness" of the figure.

Procedure Display_scale (min_x,min_y,max_x,max_y: real);

Defines the range of input co-ordinate positions that are to be visible on the display. Turtlegraphics maps the user's co-ordinates into pixel locations according to the scale specified in Display_scale.

This procedure sets the viewport (see "Figures and the Port") to encompass the whole display. The display bounds apply to input data. For the actual display, these bounds can be any values the user requires, but user-created figures always have (0,0) as their lower left-hand corner.

The default display scale is:

min_x = 0, max_x = 319
min_y = 0, max_y = 199

... which is simply the array of pixels on the full display.

As an example, if a user wishes to graph a financial chart from the years 1970 to 1980 along the x axis, and from 500,000 to 500,000,000 along the y axis, the following call could be used:

```
Display_scale(1970, 5.0E5, 1980, 5.0E8)
```

After this, calls to turtle operations could be done using meaningful numbers rather than quantities of pixels.

Function Aspect_ratio : real ;

Returns a real number that is the width/height ratio of the CRT. This can be used to compute parameters for Display_Scale that provide square aspect ratios.

If an application is designed to show information where the aspect ratio of the display is critical (e.g., circles, squares, pie-charts, etc.) it must insure that the ratio:

(max_x - min_x) / (max_y - min_y)

... is the same as the aspect ratio of the physical screen upon which the image is being displayed. When the Turtlegraphics unit is initialized, min_x and min_y are set to 0. max_x is initialized to the number of pixels in the x direction, and max_y is initialized to the number of pixels in the y direction. In order to change to different units that still have the same aspect ratio, a call similar to the following can be used:

```
Display_scale(0, 0, 100*ASPECT_RATIO, 100);
```

This utilizes Function Aspect_ratio described above, and makes the y axis 100 units long.

Turtlegraphics always treats the turtle as being in a fixed pixel location. Changing the scaling of the system with a call to this routine in the middle of a program does not alter the pixel position of any of the turtles in the figures. However, the values returned from X_pos and Y_pos may change.

Figures and the Port

The programmer can create and delete new figures, each with its own turtle. When a new figure is created, it is assigned an integer, and this integer refers to that figure in subsequent calls to Turtlegraphics procedures. New figures can be saved (Putfigure) or displayed on the screen (Getfigure).

The actual display is always referred to as figure 0.

The active portion of the display can be restricted by calling Viewport, which creates a "window" on the screen in which all subsequent graphics activity takes place. The user might create a figure, specify the port, then display that figure (or a portion of it) within the port. Specifying a viewport does not restrict turtle activity, it merely restricts what is displayed on the screen.

User-created figures can be saved in p-System disk files. See "Fotofiles".

Function Create_figure (x_size,y_size: real): integer;

Creates a new figure which is rectangular, and has the dimensions (x_size, y_size), where (0,0) designates the lower left-hand corner. The dimensions are in units of the current display scale. The figure is identified by the integer returned by Create_figure.

When a figure is created it contains its own turtle, which is at 0,0 and has a direction of 0 (it faces the right-hand side of the figure). The turtle in a user-created figure can be used by calling Activate_Turtle (described in "The Turtle").

Procedure Delete_figure (screen: integer);

Discards a previously created display figure area.

Though figures may be created and destroyed, indiscriminate use of these constructs may rapidly exhaust the memory available in the p-System due to Heap fragmentation. For example, a figure may be created using Create_figure (or it may be read in from disk using Function Load_Figure, described below). If possible, after that figure is used (for example, with a Getfigure, Putfigure, Load_figure or Store_figure operation) it should be deleted before other figures are created. If many figures are created, and randomly deleted, the Heap fragmentation problem may occur.

Procedure Getfigure (source_screen: integer; corner_x,corner_y: real; mode: integer);

Transfers a user-created figure (the "source") to the display screen (the "destination") using the drawing mode specified. The figure is placed on the display such that its lower left-hand corner is at (corner_x, corner_y). The x and y positions are specified in the units of the current display scale. If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

These are the effects of the drawing mode:

0 = Nop - does not alter the destination.

1 = Substitute - each pixel in the source replaces the corresponding pixel in the destination.

2 = Overwrite - each pixel in the source that is not of the source's background color replaces the corresponding pixel in the destination.

3 = Underwrite - each pixel in the source that is not of the source's background color is copied to the corresponding pixel in the destination only if the corresponding pixel is of the destination's background color.

4 = Complement - for each pixel in the source that is not of the source's background color, the corresponding pixel in the destination is complemented.

Values greater than 4 are treated as Nop.

If a portion of the source figure falls outside the display or the window, it is set to the source's background color.

Procedure Putfigure (destination_screen: integer; corner_x,corner_y: real; mode: integer);

Transfers a portion of the display screen to a user-created figure using the drawing mode specified (see above). The portion transferred to the figure is the area of the display that the figure covers when it is placed on the display with its lower left-hand corner at (corner_x, corner_y). If the

display scale has been modified since the figure was created, the results of this procedure are unpredictable.

Note: When a figure is moved to the display by Getfigure, further modifications to the display do not affect the copy of the figure that is saved in memory. If the user wishes to save the results of graphics work on the display, it is necessary to call Putfigure.

Procedure Viewport (min_x,min_y, max_x,max_y: integer);

Defines the boundaries of a "window" which confines subsequent graphics activities. The Viewport procedure applies only to the actual display. When a window has been defined, graphics activities outside of it are neither displayed nor retained in any way. Therefore, lines, or portions thereof, that are drawn outside the window are essentially lost and will not be displayed (this is true even if the window is subsequently expanded to encompass a previously drawn line). The viewport boundaries are specified in the units of the current display scale. If the specified size of the viewport is larger than the current range of the display, the Viewport is truncated to the display limits.

Pixels

It is possible to ascertain (Read_pixel) or write (Set_pixel) the color of an individual pixel within a given figure. These routines are more specific than the turtle-moving routines. They are less straightforward to use, but give the programmer greater control.

Function Read_pixel (screen: integer; x,y: real): integer;

Returns the value of the color of the pixel at the x,y location in the specified figure. The x,y location is specified in the units of the current Display_scale.

Procedure Set_pixel (screen: integer; x,y: real; shade: integer);

Sets the pixel at the x,y location of the specified figure to the specified color. The x,y location is specified in the units of the current Display_scale.

Fotofiles

The programmer may create disk files that contain Turtlegraphics figures. New figures may be written to a file, and old figures restored for viewing or modification.

When figures are written to a file, they are written sequentially, and assigned an index that is their location in the file. They may be retrieved "randomly" by using this index value.

The p-System name for files of figures always contains the suffix .FOTO. It is not necessary to use this suffix when calling Read_figure_file or Write_figure_file (if absent, it will be supplied automatically).

Function Read_figure_file (title: string): integer;

Specifies the title of a file from which all subsequent figures will be loaded. If a figure file is already open for reading when this function is called, it is closed before the new file is opened. Only one figure file may be open for reading at a single time. This function returns an integer value which is the IORESULT of opening the file.

Function Write_figure_file (title: string): integer;

Creates an output file into which user-created figures may be stored. If another figure file is open for writing when this function is called, it is closed, with lock, before the new file is created. Only one figure file may be open for writing at a single time. This function returns an integer result which is the IORESULT of the file creation.

Function Load_figure (index: integer): integer;

Loads the indexed figure from the current input figure file and assigns it a new, unique, figure number. An automatic Create_figure is performed. If the operation fails for any reason, a Figure_number of zero (0) is returned.

Function Store_figure (figure: integer): integer;

Sequentially writes the designated figure to the output figure file. The function returns an integer that is the figure's positional index in the current output figure file. Positional indexes start at one (1). If the index returned equals zero (0), Turtlegraphics did not successfully store the figure.

Routine Parameters

The following is the interface section for the Turtlegraphics unit, showing the parameters to all Turtlegraphics routines:

```
Unit Turtlegraphics;

Interface

Procedure Display_scale( min_x, min_y,
                         max_x, max_y: real);
Function Aspect_ratio : real;
Function Create_figure( x_size, y_size:
                         real ) : integer;
Procedure Delete_figure( screen:
                         integer );
Procedure Viewport( min_x, min_y, max_x,
                         max_y : real );
Procedure Align_cursor( x,y : real );
Procedure Fillscreen( screen:
                         integer; shade:
                         integer);
Procedure Background( screen: integer;
                         shade : integer );
Function Read_pixel( screen: integer;
                         x, y : real ) :integer;
Procedure Set_pixel( screen:integer;
                         x,y:real; shade:color );
Procedure Getfigure( source_screen:
                         integer,
                         corner_x, corner_y: real;
                         mode : integer );
Procedure Putfigure( destination_screen:
                         integer,
                         corner_x, corner_y: real;
                         mode : integer );
Function Read_figure_file( title : string ):
                         integer;
Function Write_figure_file( title : string ):
                         integer;
Function Load_figure( index : integer ):
                         integer;
Function Store_figure( figure: integer):
                         integer;
```

```
Procedure Activate_Turtle( screen:
                           integer );
Function Turtle_x : real;
Function Turtle_y : real;
Function Turtle_angle : real;
Procedure Move( distance : real );
Procedure Moveto( x,y : real );
Procedure Turn( rotation : real );
Procedure Turnto( heading : real );
Procedure Pen_mode( mode : integer );
Procedure Pen_color( shade : integer );
```

Sample Program

Here is a sample program that illustrates a number of Turtlegraphics routines:

```
PROGRAM SPIRALDEMO;


   USES IBMspecial, Turtlegraphics;

   VAR I,J,MODE: INTEGER;
         C: CHAR;
         COLOR: INTEGER;
         SEED: INTEGER;
         LX,LY,UX,UY: REAL;

   FUNCTION RANDOM(RANGE:INTEGER) : INTEGER;
         BEGIN
         SEED:= SEED*233+113;
         RANDOM:= SEED MOD RANGE;
         SEED:= SEED MOD 256;
         END;

   PROCEDURE CLEARBOTTOM;
         {clears bottom line of screen
           for prompts}
         BEGIN
         ALIGN_CURSOR(0,0);
         WRITE
           (                                  );
         ALIGN_CURSOR(0,0);
         END;
```

SEG/UNITS/LINK

```
BEGIN
CLEARBOTTOM;        {various initializations}
WRITE( ENTER RANDOM NUMBER:   );
READ(SEED);
CLEARBOTTOM;
DISPLAY_SCALE(0,0,200*ASPECT_RATIO,200);
      {Aspect__Ratio used so
        pattern will be round}
COLOR:= 0;
WRITE( ENTER VIEWPORT LL CORNER:   );
READ(LX,LY);
CLEARBOTTOM;
WRITE( ENTER VIEWPORT UR CORNER:   )
READ(UX,UY);
CLEARBOTTOM;
WRITE( PENMODE=   );
READ(MODE);

VIDEOMODE(4);
      {put display in graphics mode}
PALETTE(0);
      {0=black, 1=green, 2=red, 3=yellow}
VIEWPORT(LX,LY,UX,UY);     {create port}
PENMODE(0);
      {use blank pen while moving it}
MOVETO(100*ASPECT_RATIO,100);
      {put turtle in center of port}
      {Aspect__Ratio ensures that it will be
        correctly centered}
PENMODE(MODE);
      {set pen to selected color}
J:= RANDOM   0)+90;
      {angle by which turtle will move
        note that turtle begins facing right
        and will move counterclockwise
        (J is positive) }
FOR I:= 2 TO 200 DO
      {draw spiral in 200 segments
        of increasing length}
      BEGIN
            {cycle throught the colors}
      COLOR:= COLOR+1;
      IF COLOR > 3 THEN COLOR:= 1;
      PENCOLOR(COLOR);
      MOVE(I);
      TURN(J);
      END;
```

```
I:= CREATE__FIGURE (UX-LX, UY-LY);
      {create figure the size of the port}
PUTFIGURE(I,LX,LY,1);
      {save it; mode overwrites
        old figure (if any) }
VIEWPORT(0,0,ASPECT_RATIO*200,200);
      {re-specify viewport in
        the lower left corner}
GETFIGURE(I,0,0,1);
      {display finished spiral}
READLN;
      {clear user input buffer}
VIDEOMODE(2);
      {reset terminal to
        80*24 character mode}
END.
```

If you have further questions about this program, or Turtlegraphics in general, we suggest you experiment. Playing with graphics can be a good deal more satisfying than playing with invisible code!

NOTES

CHAPTER 6. CONCURRENT PROCESSES

Contents

Concurrent Processes 6-3
Introduction 6-3
Semaphores 6-6
Mutual Exclusion 6-8
Synchronization 6-9
Other Features 6-11

NOTES

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.

The System shares the 8086/88/87 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.

For further information on concurrent processes see the *PASCAL Reference for the UCSD p-System.*

Introduction

A process is declared exactly as a procedure would be, with the UCSD reserved word PROCESS replacing the reserved word PROCEDURE.

Examples:

```
PROCESS ZIP;
   BEGIN ... END;

PROCESS DINNER
   (var SPLIT, BLACKEYED : peas);
   begin ... end;
```

A process is started by the UCSD intrinsic START. The principal parameter passed to START is a call to a process, e.g., START(ZIP) or START(DINNER(7,234)). START also takes three optional parameters, which are explained after the following example:

```
PROGRAM DUFFER;
var PID : processid;
      I, J : integer;

PROCESS BLUE;
  begin
    .
    .
  end;

PROCESS RED ( X, Y : integer );
  begin
    .
    .
  end;

begin
  start( BLUE );
  I := 1;  J := 2;
  start( RED( I, j ) );
  start( RED( 3, 4 ), PID );
  start( RED( 5, 5 ), PID, 300 );
  start( RED( J, 1 ), PID, I+J, 10 );
  .
  .
end.
```

In the example above, program DUFFER starts processes RED and BLUE. In fact, RED is started several times. The five processes started will each run to completion, as will the main program, and the (physical) processor will share time among them. Note that the four invocations of RED result in four different versions of RED being started, each (in this example) with different parameter values.

Each invocation of a process is assigned an internal PROCESSID. PROCESSID is a UCSD predeclared type. The user may learn what processid has been assigned a given process invocation by using an optional second parameter. Thus, in START(RED(3,4), PID); the variable PID is set to a new PROCESSID value. Processids are chiefly for the use of the System and system programmers.

The optional third parameter to START is the stacksize parameter. It determines how much memory space is allocated to the process invocation (the default is 200 words).

The optional fourth parameter to START is a priority value. This determines the proportion of processor time that the process will receive before it is completed. The priorities assigned to processes are used by the System to decide which active process gets to use the available processor. Higher priority processes are given the processor 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 the *PASCAL Reference for the UCSD p-System.*

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 one (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 the *PASCAL Reference for the UCSD p-System.*

SEMINIT initializes a sempahore 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.

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 signaled. A process may synchronize with the interrupt by waiting on the semaphore.

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_emtpy );
         .
         .
         {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.

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 example on the next page, 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.
```

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 the UCSD p-System* for further details.

NOTES

CHAPTER 7. UTILITIES

Contents

Utilities 7-5
Preparing Assembly Codefiles For Use Outside of the System 7-5
Preparing Codefiles For Compression ... 7-6
Running COMPRESSOR 7-7
Action and Output Specification 7-8
Patch 7-10
EDIT Mode 7-10
TYPE Mode 7-12
DUMP Mode 7-12
A Note on Prompts 7-16
The Decoder Utility 7-16
Duplicating Directories 7-22
COPYDUPDIR 7-22
MARKDUPDIR 7-23
XREF -- The Procedural Cross-Referencer 7-24
Introduction 7-24
Referencer's Output 7-25
Lexical Structure Table 7-25
Using Referencer 7-28
Limitations 7-30
The Debugger 7-31
Invoking and Exiting the Debugger ... 7-33
Displaying and Altering Memory 7-35
Further Single-Stepping Options 7-36
Example of Debugger Usage 7-38
Summary of the Commands 7-39
The RECOVER Utility 7-41
The Tape Utility 7-43
The Print Spooler 7-44
The Native Code Generator 7-45
The TV Adjust Utility 7-48
The SETBAUD Utility 7-49
The Printer Configuration Utility 7-49

The Disk Format Utility 7-50
SETUP 7-51
Running SETUP 7-51
Miscellaneous Notes for SETUP 7-53
The Data Items in
SYSTEM.MISCINFO 7-54
BACKSPACE 7-55
CODE POOL BASE[First Word] 7-55
CODE POOL BASE[Second Word] ... 7-55
CODE POOL SIZE 7-55
EDITOR ACCEPT KEY 7-55
EDITOR ESCAPE KEY 7-55
EDITOR EXCHANGE-DELETE
KEY 7-56
EDITOR EXCHANGE-INSERT
KEY 7-56
ERASE LINE 7-56
ERASE SCREEN 7-56
ERASE TO END OF LINE 7-56
ERASE TO END OF SCREEN 7-56
HAS 8510A 7-57
HAS BYTE FLIPPED MACHINE 7-57
HAS CLOCK 7-57
HAS EXTENDED MEMORY 7-57
HAS LOWER CASE 7-57
HAS RANDOM CURSOR
ADDRESSING 7-57
HAS SLOW TERMINAL 7-57
HAS SPOOLING 7-57
HAS WORD ORIENTED
MACHINE 7-58
KEY FOR BREAK 7-58
KEY FOR FLUSH 7-58
KEY FOR STOP 7-58
KEY TO ALPHA LOCK 7-58
KEY TO DELETE CHARACTER 7-59
KEY TO DELETE LINE 7-59
KEY TO END FILE 7-59
KEY TO MOVE CURSOR DOWN ... 7-59
KEY TO MOVE CURSOR LEFT 7-59
KEY TO MOVE CURSOR RIGHT 7-59

KEY TO MOVE CURSOR UP 7-59
LEAD IN FROM KEYBOARD 7-60
LEAD IN TO SCREEN 7-60
MOVE CURSOR HOME 7-60
MOVE CURSOR RIGHT 7-60
MOVE CURSOR UP 7-60
NON PRINTING CHARACTER 7-61

PREFIXED [*itemname*> 7-61
SCREEN HEIGHT 7-61
SCREEN WIDTH 7-61
SEGMENT ALIGNMENT 7-61
STUDENT 7-61
VERTICAL MOVE DELAY 7-62

NOTES

Utilities

The UCSD p-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 p-System, yet not used frequently enough to warrant their being included among the System commands.

Preparing Assembly Codefiles For Use 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.

Preparing Codefiles For Compression

The assembly procedure(s) must be assembled with the 8086/88/87 Assembler, and linked with the Linker (see the *Assembler Reference for the UCSD p-System*, and Chapter 5 of this manual). 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 the ASSEMBLER REFERENCE for the UCSD p-System). .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.

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 <version>

Type ! to escape

Do you wish to produce a relocatable object file? (Y/N)

Unless 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 an *enter*. 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 (<ent> for same) :

Enter the name of the output file, which can be any legal filename (COMPRESSOR does not append a .CODE suffix). Typing an *enter* 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.

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, for example:

<**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, for example:

<**starting byte address of loaded code**>
+ <**byte offset**>
= <**byte address of word to be relocated**>

Each byte address pointed to by a relocation entry is a word quantity which is relocated by adding the byte address of the front of the loaded code.

Important Note: If you are relocating your file towards the high end of the 16-bit address space, you must ensure that the relocated file will not wrap around into low memory (i.e., *relocation base address* + *codefile size* must be less than or equal to FFFF(hex)). COMPRESSOR performs no internal checking for this case.

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.

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.

EDIT Mode

When PATCH is first eX(ecuted, the user is in EDIT mode. DUMP is reached by typing D. No information is lost in toggling back and forth between the two modes.

EDIT allows the user to open a file or device, read selected blocks (specified by relative block number) into an edit buffer, then either view that buffer, or

modify it (with TYPE) and write the modified block back to the file. Buffers are displayed on the screen in desired format, and edited in a manner similar to the Screen Oriented Editor.

The individual commands of EDIT are explained in some detail below. When it is impossible to perform a command, PATCH responds with self-explanatory error messages.

The promptlines for EDIT are:

EDIT : D(ump, G(et, R(ead, S(ave, M(ix,
T(ype, I(nfo, F(or, B(ack, ?

EDIT : V(iew, W(ipe, Q(uit, ?

D(ump - calls DUMP.

G(et - opens the file or device that one wishes to use, and reads block zero into the buffer.

R(ead - reads a specified block from the current file.

S(ave - writes the contents of the buffer out to the current block.

M(ixed - changes the display format for the current block. Typing M toggles between the two formats mixed and hex. 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).

TYPE Mode

TYPE, like the Screen Oriented Editor, allows the information on the screen to be modified by moving the cursor around and typing over existing information. If you make errors while using TYPE, do not S(ave the buffer while in EDIT mode, but R(ead the block over and try again.

The promptline for TYPE is:

**TYPE : C(har, H(ex, F(ill, U(p, D(own, L(eft,
R(ight, <vector arrows>, Q(uit**

C(haracter - exchanges bytes in the buffer for ASCII characters as they are typed, starting from the cursor and continuing until an *etx* is typed. Only printable characters are accepted.

H(ex - exchanges bytes in the buffer for hex digits as they are typed, starting from the cursor and continuing until a Q is typed. (Hex digits can be either upper or lower case.)

F(ill - fills a portion of the current block with the same byte pattern. Accepts either ASCII characters

or hexadecimal digits for the pattern. When finished, the cursor will be positioned after the last byte filled.

The following commands move the cursor around within the block of data being displayed. The cursor is always at a particular byte. Rather than moving off the screen, the cursor wraps around from side to side and from top to bottom.

U(p - moves the cursor up one row.

D(own - moves the cursor down one row.

L(eft - moves the cursor left one column.

R(ight - moves the cursor right one column.

vector arrows - these are the vector arrows as used in the Screen Editor. They will do the same respective actions as U,D,L,R.

Q(uit - quits the TYPE mode and returns to the EDIT mode.

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 a 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.

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.)

The Decoder Utility

The Decoder utility is called DECODE.CODE. 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 be decoded only 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 *enter*), 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 *enter* 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   I:INTEGER;
 4     1:d  2
 5     1:d  2  SEGMENT PROCEDURE ADDI;
 6     1:0     BEGIN
 7  3  1:1  0    I:=I+I;
 8  3  1:0  5  END;
 9  3  1:0  7
10  2  1:0  0  BEGIN
11  2  1:1  0    I:=50;
12  2  1:1  4    REPEAT
13  2  1:2  4      ADDI;
14  2  1:1  7    UNTIL I=400;
15      :0 14  END.
```

... Decoder would prompt for input and output filenames. Then, if D(ictionary was typed, the following would be displayed:

```
INDEX NAME  START SIZE  VERSION  M_TYPE     SG# SEG_TYPE  RL FMY_NAME or
                                                             DSIZE      SGRF  HISG
0     DEMO  2     20    IV.0     M_PSELDO   2   PROG_SEG  R  1           10     4
1     ADDI  1     14    IV.0     M_PSELDO   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
```

Figure 7-1. Dictionary Entry.

... and Decoding A(ll of this program would produce the following disassembly:

```
DATA POOL:   SEGMENT DEMO   PROCEDURE  1  BLOCK  2  BLOCK OFFSET  0

  0:0013.    0000.   4544.ED  4F4D.OM    2020.    2020.    001E.    0000.
                   block #  2  offset in block  16
  OFFSET                                 HEX CODE
   0(000):   LDCB     50                 8032
   2(002):   SRO       1                 A501
   4(004):   CXG       3        1        940301
   7(007):   SLDO      1                 30
   8(008):   LDCI    400                 819001
  11(00B):   EQUI                        B0
  12(00C):   FJP       4                 D4F6
  14(00E):   CXG       4        2        940402
  17(011):   RPU       0                 9600

DATA POOL:   SEGMENT ADDI   PROCEDURE  1  BLOCK  1  BLOCK OFFSET  0

  0:000d.    0000.   4441.DA  4944.ID    2020.    2020.    0015.    0000.
                   block #  1   offset in block  16
  OFFSET                                 HEX CODE
  0(000):    SLDO      1                  30
  1(001):    SLDO      1                  30
  2(002):    ADI                          A2
  3(003):    SRO                          A501
  5(005):    RPU                          9600
```

Figure 7-2. Disassembled Output.

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 *enter*; e.g., 0 *enter* 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 be designated M_8086.

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 for the UCSD p-System.*

The disassembled code itself is displayed in blocks. The OFFSET column shows the offset in bytes from the front of the procedure (the count is in both decimal and hex). Then the P-code mnemonic is displayed, followed by the operands, if any, and finally the HEX CODE for that particular instruction.

The OFFSET column corresponds to the fourth column in a compiled listing.

Jump operands are displayed as offsets relative to the start of the procedure, rather than IPC-relative (IPC = Interpreter program counter). This is to mak

UTILITIES

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 *enter* 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(I : 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

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.

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.

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 121 15-Nov-81  6 Datafile
   .
   .
   .
OR

<unused>        4            6
SYSTEM.PASCAL 121 15-Nov-81  10 Datafile
   .
   .
   .
```

... both of the above cases indicate disks that have not been marked. Below is the directory of a properly marked disk:

```
SYSTEM.PASCAL 121 15-Nov-81  10 Datafile
   .
   .
   .
```

To execute this program, X(ecute 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.

XREF -- The Procedural Cross-Referencer

Introduction

The Procedural Cross-Referencer was developed from an original cross-referencer written by Arthur Sale. It 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-non-local variable references. It thus provides information about the inter-procedural dependencies of a program.

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.

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.

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 its 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.) This list is in order of occurrence in the text; a procedure is not listed more than once.

The Procedure Call Table

This is a list of procedures, in alphabetical order, and, for each procedure, the procedures which call it.

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.

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.

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.

Using Referencer

When the Referencer is run, it should be on the same disk as the program it is analyzing.

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 *enter*. Typing an empty filename (*enter*) exits Referencer.

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.

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.

The Debugger

This section describes the Debugger utility. The Debugger can be used as an aid to debugging compiled programs. It can be invoked from the main System promptline, or during the execution 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. If a space is typed for any of these, the command is exited. The exception to this is the Breakpoint command as described below. If an inappropriate response is given to any prompt (such as proc Num? 3000) then the response will not be accepted.

In 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 Internal Architecture Guide. It is possible to cause the System to die if the Debugger is used incorrectly. Also, in order to use the Debugger, it is useful to have a compiled listing of the program being debugged. This listing is helpful in determining P-code offsets etc., and should be current.

The Debugger can be used more easily if the code being debugged is compiled with the $D+ option (the *PASCAL Reference for the UCSD p-System* describes compiler options in detail). The $D option (which defaults to $D-) instructs the compiler to output symbolic debugger information for those portions of a program which are compiled with $D+ turned on. Variables within a given routine may be specified by name (rather than data segment offset number) if at least one statement within that routine is compiled $D+. Breakpoints may be specified by line number (rather than P-code offset number) for all statements covered by the $D+ option. Once a program is debugged, however, it should be recompiled without symbolic debugger information. This is because that information increases the size of the codefile.

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. If 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. If the Debugger is exited using the Q(uit option, it will be disabled. If it is later re-invoked, it will be in a fresh state. If the Debugger is exited using the R(esume option, execution will continue from where it left off and the Debugger will still be active. If it is then re-invoked, it will be in a non-fresh state. If 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. If, 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 Procname or #? appears: the number of the desired procedure or the first eight characters of the valid procedure name should be typed followed by *space*. If a procedure number is entered, then Offset #? appears: the desired offset within the procedure should be typed followed by *space*. If a procedure name is entered after the Procname or # prompt, the following line will be displayed: First#__ last#__ Line#?. The underlines will actually be numbers which indicate the first and last line numbers. The desired line number within the specified range should be entered. (Note: The "First #__ Last #__" information is displayed only if symbolic debugger information is found). 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.

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, E(xtended or P(rocedure. If G or L is typed, the prompt Varname or Offset#? appears: the desired offset into the data segment or variable name should by typed. (Note: Only Offset # appears if symbolic debugger information cannot be found.) If I is typed, Delta Lex Level? is also prompted (when an offset number is input). If E is typed, the prompts Seg # and Offset # are displayed (extended variables may not be specified symbolically). If P is typed, an offset within a specified procedure may be displayed: Segment name?, Procname or #? and Varname or Offset #? are all prompted in sequence.

When any of the non-symbolic options are used, a line similar to the following is displayed:

```
( 1) S=INIT  P#1  VO#1 2C1A: OB 05 53 43
     41 4C 43 61  -SCALCa
```

... in this example, a Local (l) segment of memory is displayed. The segment is INIT, procedure 1, variable offset 1 at absolute hex location 2C1A. Following this, eight bytes are displayed, first in HEX and then in ASCII (a - indicates that the character is not a printable ASCII character).

If the desired variable had been entered symbolically, the same line might appear:

```
( l) S=INIT  P=FILLTABL  V=TABLE1  2C1A:
     OB 05 53 43 41 4C 43 61  -SCALCa
```

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. When altering in hex mode, any characters which are to be left unchanged may be skipped by typing *space*. In the ASCII mode, any characters to be left unchanged may be skipped by typing *enter.*

A text file may be viewed from the debugger by typing F(ile. The Filename? First line #? and Last line #? prompts are then displayed. This command will list as many lines as possible in the window between first line and last line of the indicated file.

Further Single-Stepping Options

When the single-stepping mode (described in "Invoking and Exiting the Debugger") 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?, Procname or #?, and either First #__ Last #__ Start Line #?, End Line #? or Start Offset #?, End Offset #?. This command

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.

Example of Debugger Usage

Suppose the following program is to be debugged:

```
Pascal Compiler IV.0

1 0  0:d1 {$L LIST.TEXT}
2 2  1:d1 PROGRAM NOT_DEBUGGED;
3 2  1:d1 VAR I,J,K:INTEGER;
4 2  1:d4       B1,B2:BOOLEAN;
5 2  1:00 BEGIN
6 2  1:10     I:=1;
7 2  1:13     J:=1;
8 2  1:16     IF K<> 1 THEN WRITELN
                      ('Whats wrong?');
9 2   :00 END.

End of Compilation.
```

First we enter the Debugger and set a breakpoint at the beginning of the IF statement:

```
(BS) Set break #? 0 Segname? NOTDEBUG
       Procname or #? 1 Offset #? 6
(EP)
(R)
```

After setting the 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 that 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.

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.

Command	Description
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 segment of given procedure.
E(xtended	Displays variables 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(ddress	Displays a given address.
P(code	Disassembles a given procedure.
M(emory	
L(ock	Memlocks the Debugger.
S(wap	Memswaps the Debugger.
F(ile	Allows viewing of text files.

The RECOVER Utility

RECOVER is a utility which attempts to recreate the directory of a disk whose directory has accidentally been destroyed. It is shipped as RECOVER.CODE on the Utility diskette.

RECOVER displays several yes/no prompts. These must be answered with upper-case letters: lower-case letters are ignored.

Following is a list of RECOVER's prompts, with a description of appropriate responses:

RECOVER - Version IV.0.9
ENTER TODAY'S DATE MM-DD-YY

... the user should enter a valid date, followed by *enter*. 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 *enter*).

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. On the IBM Personal Computer, each disk contains 320 blocks.

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)?

... an 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.

The Tape Utility

The tape utility is called TAPE.CODE. It is used to transfer a file between cassette tape and disk. When TAPE is eX(ecuted it will ask whether a transfer is to be made to tape or from tape with the following prompt:

R(ead from tape, W(rite to tape, Q(uit

It will then prompt for the file name:

Filename?

This file name will either be the file to be transferred from disk to tape, or it will be the name the file will be given when it is read from tape and placed on disk. At most one file may reside on a cassette tape.

The Print Spooler

The utility SPOOLER.CODE makes use of the Operating System unit SPOOLOPS. Within this unit there is a process called Spooltask. Spooltask is started at boot time and runs concurrently with the rest of the UCSD p-System. Spooltask looks for a one block file on disk called SYSTEM.SPOOLER. This file, if it exists, is a queue of filenames. Spooltask will send these files to the Printer as the user is running the p-System normally.

Utility SPOOLER.CODE interfaces with SPOOLOPS and uses routines within it to generate and alter the print queue within SYSTEM.SPOOLER. When SPOOLER is eX(ecuted, the following promptline appears:

Spool: P(rint, D(elete, L(ist, S(uspend, R(esume,
A(bort, C(lear, Q(uit

P(rint will prompt for the name of the file to be printed. This name will then be added to the queue. If SYSTEM.SPOOLER does not exist, it will be created. In the simplest case, P(rint may be used to send a single file to the printer. At most 21 files may be placed in the print queue.

D(elete will prompt for a file name to be taken out of the print queue within SYSTEM.SPOOLER. All occurrences of that file name will be taken out of the queue.

L(ist displays the files currently within the queue.

S(uspend temporarily halts the printing of the current file.

R(esume continues the printing of the current file after a S(uspend. R(esume also starts printing the next file in the queue after an error or an A(bort.

A(bort permanently stops the printing process of the current file and takes it out of the queue.

C(lear deletes all file names from the queue.

Q(uit exits the SPOOLER utility and starts transferring files to the printer.

If an error occurs (e.g. a nonexistent file is specified in the queue), the error message will appear only when the p-System is at the main system promptline. If necessary, SPOOLOPS will wait until the user returns to the outer level.

Program I/O to the printer may run concurrently with printer I/O by SPOOLOPS. SPOOLOPS will finish the current file and then turn the printer over to the user program. (The user program will be suspended as it waits for the printer.) The user program should only do Pascal (or other high level) writes to the printer. If the user program does printer I/O using UNITWRITE, the I/O will be sent immediately and will be randomly interspersed with the I/O going on in the background.

The Native Code Generator

The Native Code Generator is called CODEGEN.CODE. It inputs an executable codefile and produces another executable codefile. The output file, however, contains a mixture of P-code and 8086/88/87 Native Code (N-code). The Code Generator selectively translates embedded sections of the P-code input file into equivalent N-code.

Generally, N-code executes much more quickly than P-code, but requires more memory space. Time critical code may take fuller advantage of the processor's speed if it is translated into N-code.

The selection of which code is translated into native code has been left under user control. The user specifies what code he wishes translated to N-code by enclosing the desired sections with the compile-time switches $N+ and $N-. When the compiler encounters the $N+ option, it will begin emitting additional P-codes containing information necessary for the Code Generator to perform its translation. When it encounters the $N- options, it will discontinue generation of the additional P-codes. The default setting for this compiler option is $N-.

An entire routine (Procedure, Function, etc.) is the smallest unit that can be translated into Native Code at one time (in the current implementation). The $N+ must occur before the first BEGIN within that routine. The Code Generator will not generate any Native Code unless at least one entire routine is included between a $N+ and the corresponding $N-.

The object code produced by the compiler from source containing the $N+ option is executable like any other P-code file. The only difference is a slight increase in codefile size due to the extra P-code hooks placed there for possible code generation.

If there are any references to assembly language routines within a codefile, these routines must be linked in before that codefile may be processed through the Code Generator (see the *Assembler Reference for the UCSD p-System* manual for information about linking assembly language routines into codefiles).

The Code Generator will not necessarily translate all P-codes within the section(s) of code specified by the user into N-code. Due to the unwieldy nature of their machine code equivalents, some P-codes will be left in their original P-code form. Also, only those sections of P-code which encompass an entire Routine (Procedure, Function or Process in Pascal) will be translated.

The Code Generator will accept codefiles generated by any of the compilers released as part of the UCSD p-System. This includes the Pascal and FORTRAN compilers. The Code Generator will only fail to generate an output file if the input file is not a valid executable codefile.

The Code Generator will produce an object codefile whose execution behavior is identical to that of the input codefile, except for differences with respect to execution speed, object code size, or implementation dependencies. An example of an implementation dependency might involve conditional expression evaluation. If a loop were constructed in the source program that checked the value of a variable and used the variable to index into an array in the same expression, it would be possible for the P-code object file to give a value-range error for some value of the variable. The corresponding N-code however, might short-circuit the array indexing and subsequent value-range check if the first test failed.

Concurrency is implemented in such a way that P-codes are uninterruptable operations. If a p-machine interrupt occurs during the execution of a P-code, the event is queued until the P-code finishes. In this respect, any embedded N-code in the code file behaves as if it were a single P-code. It is the case that if the user has any assembly language routines bound into a codefile, then the execution of an entire routine appears to the p-machine to be a single P-code. Any p-machine interrupts occurring during the execution of the assembly routine would be queued until the end of the routine, or the beginning of the next P-code. Since the Code Generator does not, in general, translate all P-codes in the selected sections of the input file into N-code, it is typically a much smaller sequence of N-code that appears as a single P-code to the p-machine. If, however, the user desires to force more frequent or specific limits on the size of these

N-code sequences (to allow for more frequent event checking), the $N- options followed immediately by the $N+ option will enforce a brief return to P-code.

Finally, the Code Generator has an option to give an assembly language format listing of all routines for which any translation has been performed.

The TV Adjust Utility

Utility TVADJUST modifies the video parameters of the IBM Personal Computer to compensate for differences among commercial TV consoles. Depending on the manufacturer and the condition of the TV set, some of the first three columns of characters may not be visible when the Personal Computer is connected to it. To correct the problem the TVADJUST utility may be run. When executed it displays a column ruler across the display. Hitting the left and right cursor control arrows reposition the display relative to the TV screen. Typically the user should type the right arrow until the first column becomes visible. If the display is moved too far right, the left arrow can be used to move it back. Once the display is properly positioned, the *enter* key will terminate execution and save the new display parameters. The parameters remain in memory until the system is powered off or re-booted. If your TV set requires the use of the TVADJUST utility it will be necessary to execute it every time the system is booted.

If the display is positioned too far to the left, the synchronization of the display may be lost and the picture can start to roll. To correct this problem, move the display right until the rolling ceases.

The SETBAUD Utility

The SETBAUD utility sets the baud rate of the REMIN and REMOUT ports in bits per second. When the utility is run, it displays a list of the available baud rates. You select the baud rate by typing the integer associated with the desired rate and then hitting *enter*.

The Printer Configuration Utility

The PRNCONFIG utility sets the various printer options available on the IBM Personal Computer printer. The options are:

- H)orizontal Tabs
- V)ertical Tabs
- C)hars/Line
- ? (displays remaining prompts)
- L)ines/Inch
- F)orm Length
- Q)uit

H)orizontal and V)ertical Tabs are set by first typing "H" or "V" and then responding to the next prompt by typing in a Tab List. A Tab List is a list of integers (representing columns or rows) separated by commas.

If "C" for C)haracters per line is typed, two options will be displayed:

A) 80 columns, B) 132 columns

"A" or "B" should be typed to select one of these.

If "L" is typed, the L)ines per inch may be chosen from among the following options:

A) 6 lpi, B) 8 lpi, C) 10 lpi

"A", "B", or "C" should be typed.

If "F" is typed, the number of lines per page may be set by responding to the prompt:

Enter lines per page:

Type in the desired number followed by *enter*.

"Q" will quit the Printer Configuration Utility and return to the main p-System promptline.

The Disk Format Utility

The DISKFORMAT Utility formats a disk so that it may be used by the UCSD p-System on the IBM Personal Computer. When it is executed the following promptline is displayed:

Enter unit number of disk to be formatted (4..5)

"4" or "5" should be typed indicating drive #4: or #5:. If you type "5", the next prompt is:

Insert disk in unit 5 and press <enter>...

When this is done, the formatting process, which takes several seconds, will begin. This utility does not create a p-System directory on the newly formatted disk. To create a directory, the Z(ero command of the Filer should be used.

SETUP

SETUP is provided as a System utility (on the Utility disk) called SETUP.CODE. SETUP changes a file that contains details about your terminal, and a few miscellaneous details about the System in general. SETUP can be run, and the data changed, as many times as you desire. After running it, it is important to reboot (or I(nitialize) so that the System will start using the new information. It is also important to back up old data.

The file that SETUP uses to store all of this information is called SYSTEM.MISCINFO. Each System initialization loads it into main memory. New versions of SYSTEM.MISCINFO are created by SETUP, and are called NEW.MISCINFO. Backups are created by renaming or copying SYSTEM.MISCINFO with the Filer.

SYSTEM.MISCINFO contains three types of information:

1) Miscellaneous data about the System.

2) General information about the terminal.

3) Specific information about the terminal's various control keys.

Running SETUP

SETUP is a utility program, and is run like any other compiled program: type X for eX(ecute, and then answer the prompt with SETUP *enter*. It will display the word INITIALIZING followed by a string of dots, and then the prompt:

SETUP: C(HANGE T(EACH H(ELP
Q(UIT [version]

To invoke any command, just type its initial letter.

H(ELP gives you a description of the commands that are visible on any promptline where it appears.

T(EACH gives a detailed description of the use of SETUP. Most of it is concerned with input formats. They are mainly self-explanatory, but if this is your first time running SETUP, you should look through all of T(EACH.

C(HANGE gives you the option of going through a prompted menu of all the items, or changing one data item at a time. In either case, the current values are displayed, and you have the option of changing them. If this is your first time running SETUP, the values given are the system defaults.

Q(UIT has the following options:

- H(ELP),
- M(EMORY) UPDATE, which places the new values in main memory,
- D(ISK) UPDATE, which creates NEW.MISCINFO on your disk for future use,
- R(ETURN), which lets you go back into SETUP and make more changes, and
- E(XIT), which ends the program and returns you to the System promptline.

Please note that if you have a NEW.MISCINFO already on your disk, D(ISK) UPDATE will write over it.

"Miscellaneous Notes for SETUP" contains a detailed description of the data items in SYSTEM.MISCINFO.

If you use SETUP to change your character set, don't underestimate the importance of using keys you can easily remember, and making dangerous keys like BREAK and ESCAPE hard to hit.

Once you have run SETUP, you should always backup SYSTEM.MISCINFO under some other name (e.g. OLD.MISCINFO), then change the name of NEW.MISCINFO to SYSTEM.MISCINFO and reboot or I(nitialize. It is indeed possible to update to memory alone, and go on using the System without rebooting, but the results may not always be what you wanted, and the backup security is more risky. In general, M(EMORY) UPDATE is a Q(UIT option that you will use only when experimenting. If you do get into a bind, remember that the current in-memory SYSTEM.MISCINFO can be saved by running SETUP and doing a D(ISK) UPDATE before you change any data items.

When you reboot or I(nitialize, the new SYSTEM.MISCINFO will be read into main memory and its data used by the System, provided it has been stored under that name on the System disk (the disk from which you boot).

Miscellaneous Notes for SETUP

The STUDENT bit, one of SYSTEM.MISCINFO's data items, should always be set to FALSE.

The HAS 8510A bit is always FALSE.

HAS WORD ORIENTED MACHINE is always FALSE.

HAS BYTE FLIPPED MACHINE is FALSE.

SETUP and the Manual refer to PREFIXED [DELETE CHARACTER]. This refers to the backspace function: read it as PREFIXED [BACKSPACE]. It will be FALSE.

Your terminal should be set to run in full duplex, with no auto-echo.

Don't use terminal functions that do a "Delete and close up" on lines or characters -- not all terminals have these functions, and so they are supplied through the Screen Oriented Editor's software.

In general, if SETUP prompts for a feature that your terminal does not have, set the item to NUL (zero).

The Data Items in SYSTEM.MISCINFO

The information in this section is very specific, and you may skip it on first reading. If you have a question about a certain data item, look in this section. Default values are shown, and sometimes our recommendations. The items are ordered according to SETUP's menu.

Please note that SETUP frequently makes a distinction between a character which is a key on the keyboard, and a character which is sent to the screen from the UCSD p-System.

There are a few characters which you cannot change with SETUP. These are CARRIAGE RETURN (*enter*), LINE FEED (*lf*), ASCII DLE (Ctrl-P), and TAB (Ctrl-I). ASCII DLE (data link escape) is used as a blank compression character. When sent to an output textfile, it is always followed by a byte containing the number of blanks which the output device must insert. If you try to use Ctrl-P for any other function, you will run into trouble.

BACKSPACE

When sent to the screen, this character should move the cursor one space to the left. Default: ASCII BS.

CODE POOL BASE[First Word]

Default is 0. (See "Extended Memory" in the Operations Guide (Part 1) of this manual.)

CODE POOL BASE[Second Word]

Default is 0.

CODE POOL SIZE

Default is 32767. (See "Extended Memory" in the Operations Guide (Part 1) of this manual.)

EDITOR ACCEPT KEY

This key is used by the Screen Oriented Editor. When pressed, it ends the action of a command, and accepts whatever actions were taken. Default: ASCII ETX (Ctrl-C).

EDITOR ESCAPE KEY

This key is used by the Screen Oriented Editor. It is the opposite of the EDITOR ACCEPT KEY - when pressed, it ends the action of a command, and ignores whatever actions were taken. Default and Suggested: ASCII ESC (Ctrl-[).

EDITOR EXCHANGE-DELETE KEY

This key is also used by the Screen Oriented Editor. It operates only while doing an eX(change, and deletes a single character. Default: ASCII p.

EDITOR EXCHANGE-INSERT KEY

Like the EDITOR EXCHANGE-DELETE KEY, this only operates while doing an eX(change in the Screen Oriented Editor: it inserts a single space. Default: ASCII o.

ERASE LINE

When sent to the screen, this character erases all the characters on the line that the cursor is on. Default: ASCII L.

ERASE SCREEN

When sent to the screen, this character erases the entire screen. Default: ASCII E.

ERASE TO END OF LINE

When sent to the screen, this character erases all characters from (and including) the current cursor position to the end of the same line. Default: ASCII K.

ERASE TO END OF SCREEN

When sent to the screen, this character erases all characters from (and including) the current cursor position to the end of the screen. Default: ASCII J.

HAS 8510A

Is always FALSE.

HAS BYTE FLIPPED MACHINE

Is always FALSE.

HAS CLOCK

Default is FALSE.

HAS EXTENDED MEMORY

Default is FALSE. (See "Extended Memory" in the *Operations Guide* (Part 1) of this manual.)

HAS LOWER CASE

Default is TRUE.

HAS RANDOM CURSOR ADDRESSING

Is always TRUE unless your terminal is not a CRT.

HAS SLOW TERMINAL

May be TRUE or FALSE. When this bit is TRUE, the system's promptlines and messages are abbreviated. It is suggested that you leave this set at FALSE. Default: FALSE.

HAS SPOOLING

Default is FALSE.

HAS WORD ORIENTED MACHINE

Is always FALSE.

KEY FOR BREAK

When this key is pressed while a program is running, the program will terminate with a runtime error. Default: ASCII NUL (Ctrl-@).

KEY FOR FLUSH

This key may be pressed while the System is sending output (writing to the file OUTPUT). The first time it is pressed, output is no longer displayed, and will be ignored ("flushed") until FLUSH is pressed again. This can be done any number of times; FLUSH functions as a toggle. Note that processing continues while the output is ignored, so using FLUSH causes output to be lost. Default and suggested: ASCII ACK (Ctrl-F).

KEY FOR STOP

This key may be pressed while the System is writing to OUTPUT. Like FLUSH, it is a toggle. Pressing it once causes output and processing to stop, pressing it again causes output and processing to resume, and so on. No output is lost; STOP is useful for slowing down a program so the output can be read while it is being sent to the terminal. Default and suggested: ASCII DC3 (Ctrl-S).

KEY TO ALPHA LOCK

This character, when sent to the screen, locks the keyboard in upper case (alpha mode). Default: ASCII DC2 (Ctrl-R).

KEY TO DELETE CHARACTER

Deletes the character where the cursor is, and moves cursor one character to the left. Default and suggested: ASCII BS (BACKSPACE).

KEY TO DELETE LINE

Deletes the line that the cursor is currently on. Default and suggested: ASCII DEL (Ctrl-BACKSPACE).

KEY TO END FILE

Sets the intrinsic Boolean function EOF to TRUE when pressed while reading from the System input files (either KEYBOARD or INPUT, which come from device CONSOLE:). Default and suggested: ASCII ETX (Ctrl-C).

KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
KEY TO MOVE CURSOR UP

These keys are recognized by the Screen Oriented Editor, and are used when editing a document to move the cursor about the screen. We suggest using the arrows on the number pad for these functions. Default (in order): ASCII B, ASCII D, ASCII C, ASCII A.

LEAD IN FROM KEYBOARD

Pressing certain keys generates a two-character sequence. The first character in these cases must always be a prefix, and must be the same for all such sequences. This data item specifies that prefix. Note that this character is only accepted as a lead in for characters where you have set PREFIXED[*itemname*] to TRUE. Default: ASCII DC1 (Ctrl-Q).

LEAD IN TO SCREEN

Some terminals require a two-character sequence to activate certain functions. If the first character in all these sequences is the same, this data item can specify this prefix. This item is similar to the one above. The prefix is only generated as a lead in for characters where you have set PREFIXED[*itemname*] to TRUE. Default: ASCII ESC (Ctrl-[).

MOVE CURSOR HOME

When sent to the terminal, moves the cursor to the upper left hand corner of the screen (position (0,0)). Default: ASCII H.

MOVE CURSOR RIGHT

When sent to the terminal, moves the cursor nondestructively one space to the right. Default: ASCII C.

MOVE CURSOR UP

When sent to the terminal, moves the cursor vertically up one line. Default: ASCII A.

NON PRINTING CHARACTER

The character that will be displayed on the screen when a non-printing character is typed or sent to the terminal while using the Screen Oriented Editor. Default and suggested: ?.

PREFIXED [<itemname>]

If any two-character sequence must be generated by a key or sent to the screen, the System will recognize that if you set PREFIXED[*itemname*] to TRUE. See the explanations for LEAD IN FROM KEYBOARD and LEAD IN TO SCREEN.

SCREEN HEIGHT

The number of lines in your display screen, starting from 1. Default: 25 (base ten).

SCREEN WIDTH

The number of characters in one line on your display, starting from 1. Default: 80 (base ten).

SEGMENT ALIGNMENT

Default 16 (base ten).

STUDENT

Should always be FALSE.

VERTICAL MOVE DELAY

May be a decimal integer from 0 to 11. Many terminals require a delay after vertical cursor movements. This delay allows the movement to be completed before another character is sent. This data item specifies the number of nulls that the System sends to the terminal after every CARRIAGE RETURN, ERASE TO END OF LINE, ERASE TO END OF SCREEN, CLEAR SCREEN, and MOVE CURSOR UP. Default: 0.

APPENDIXES

Contents

Appendix A. Error Messages A-1
Appendix B. I/O Results B-1
Appendix C. Device Numbers C-1
Appendix D. ASCII Chart D-1
Appendix E. Special Keys E-1

NOTES

APPENDIX A. ERROR MESSAGES

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.

NOTES

APPENDIX B. I/O RESULTS

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Device Number
3	Invalid 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	Invalid block number
18	Invalid buffer

NOTES

APPENDIX C. DEVICE NUMBERS

Device Number	Volume Name
0	<for System use>
1	CONSOLE:
2	SYSTERM:
3	
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 3 - Files and Filehandling. REMIN: and REMOUT: are often set to the same bidirectional port.

NOTES

APPENDIX D. 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

NOTES

APPENDIX E. SPECIAL KEYS ON ON THE IBM PERSONAL COMPUTER KEYBOARD

The IBM Personal Computer Keyboard has several special keys, and key combinations which perform various tasks. Some of these keys are used for p-System functions. Others are set aside for user defined functions. There are also screen related keys which affect the display on the CRT.

p-System-Related Keys

The following table lists the special keys that are used by the UCSD p-System on the IBM Personal Computer:

FUNCTION	KEY
ESC	ESC
DEL	Ctrl-BACKSPACE
ENTER	The bent left arrow key
EOF	Ctrl-C
BACKSPACE	The left arrow at the upper right of the keyboard
ETX	Ctrl-C
TAB	The left/right arrow key at the upper left of the keyboard

SPEC KEYS

BREAK	Ctrl-BREAK - This key is a "hard" break which forces an immediate halt in program execution, followed by System re-initialization
BREAK	Ctrl-@ - This key is a "soft" break which causes a halt in program execution at the next I/O operation, followed by System re-initialization
STOP	Ctrl-S - Stops execution at the next I/O operation until pressed again
DC1	Ctrl-Q - In Editor's I(nsert mode, pressing this twice jumps to left margin
LF	Ctrl-RETURN
FLUSH	Ctrl-F - Discards output waiting to be displayed
INS	INS - Inserts a blank character in the Editors eX(change mode
DEL	DEL - Deletes a character in the Editors eX(change mode

User-Defined Keys

The keys labeled F1 through F10 at the left of the keyboard return the following code sequences:

Function Key	Code Sequence
F1	DC1 a
F2	DC1 b
F3	DC1 c
F4	DC1 d
F5	DC1 e
F6	DC1 f
F7	DC1 g
F8	DC1 h
F9	DC1 i
F10	DC1 j

It is as if the indicated code sequences were sent to the console when one of these function keys is typed. A user program may assign whatever meaning to these keys that is desired. These special function keys may be set to return string values instead of the indicated code sequences. This can be done using the Setkeys procedure within the IBMSPECIAL unit (see Chapter 6). When this is done, typing one of these keys will actually echo a string to the console.

The following keys on the number pad represent functions 11 through 16 and may also be defined by procedure Setkeys:

11	Home
12	PageUp
13	End
14	PageDown
15	INS
16	DEL

CRT-Related Keys

The PrtSc key will send whatever is currently displayed on the CRT to the Printer.

When the screen is in 40 character mode Ctrl-Right Arrow shifts the window 20 characters to the right. Ctrl-Left Arrow shifts 20 characters to the left. These commands cause the screen to wrap around if the right-most (or left-most) portion of the screen is already displayed. Also, in 40-character mode, the usual cursor moving keys (Left Arrow, Space Bar, etc.) shift the display window one column at a time whenever the cursor moves off the screen.

For color monitors, ALT-C is a toggle that clears the screen and turns color on and off.

INDEX

Note: In the following index, PR refers to the *PASCAL Reference for the UCSD p-System*, AR refers to the *Assembler Reference for the UCSD p-System*, and AG to the *Internal Architecture Guide for the UCSD p-System.* Users interested in FORTRAN should refer to the *FORTRAN-77 Reference for the UCSD p-System.*

Boldface indicates the principal description of an item.

A

A(djust 4-16
array PR
ASCII D-1
A(ssemble 2-17
assembled code and assemblers **AR**, 1-6, 1-21, 2-17, AG
ATTACH **PR**

B

bad blocks 1-13, **3-13, 3-37**
B(ad blocks **3-13**
IBMSPECIAL 5-33
BIOS 1-17, **AG**
block-structured device 1-15, 3-5, 3-23, 3-38
BLOCKREAD **PR**
BLOCKWRITE **PR**
bootstrap 1-10, 1-12, 2-20
booting 2-3

C

CASE statement **PR**
CHAIN 1-17, **2-14**
C(hange **3-14**
CLOSE **PR**
code segment see segment
codefile 1-10, 1-18, 2-16, 2-17, **AG**
COMMANDIO 5-33
commands 1-4, **2-16, 3-12, 4-16**
comments **PR**
C(ompile 1-6, **2-18**
compiled listing **PR**
Compiler 1-18, 2-16, **PR**
compile-time options **PR**
COMPRESSOR 7-5
CONCAT **PR**
concurrent processes 6-3, **PR**
conditional assembly **AR**
conditional compilation **PR**
CONSOLE: 1-16, 2-11, 3-3, 3-23, 3-38, PR

C(opy 4-19
COPY **PR**
COPYDUPDIR 7-22
cross-referencer 7-24
cursor 4-4, 4-7, 4-13

D

D(ate **3-17**, AG
D(ebug 7-33
DEBUGGER 7-33
DECODER 7-16
default disk (*) 1-15, 2-8, 3-38
D(elete 4-9, 4-20
DELETE **PR**
device numbers 1-16, 3-38
devices 1-14
directives **PR**
directory 1-15, 2-19, 3-11, 3-18, **AG**
disks see floppy disks
Disk Format Utility 7-50
diskette, dual 4-7
DISPOSE PR
DLE 3-4
dual-sided diskette 4-7

E

E(dit 2-19
Editor 4-3
EOF **PR**
EOLN **PR**
eX(amine 3-39
EXCEPTION 1-17, 2-8, **2-13**
eX(change 4-44
eX(ecute 2-5, 2-7, 2-25
execution errors A-1, AG
execution option strings 2-7
EXIT **PR**
E(xtended list 3-18
EXTERNAL 1-21, **PR**
external routines 1-21, AG, **AR**

F

file 1-9, **3-3**, **PR**, AG
file-handling **3-3**, AG
filenames 1-10, 1-12, 2-16, 3-3, **3-7**, 4-5
F(ile 1-6, **2-19**
Filer 1-6, 1-13, 3-3, **3-8**
FILLCHAR **PR**
F(ind 4-15, **4-22**
floppy disks 1-9, 2-6, 3-13, 3-38, 7-22, 7-43
FUNCTION see routine

G

G(et 1-13, 3-6, **3-20**
GET **PR**
GOTO **PR**
GOTOXY **PR**

H

H(alt **2-20**
HALT **PR**
heap HB, **AG**

I

IBMSPECIAL 5-33
IMPLEMENTATION 1-21, 5-7, **5-8**, 5-13

$Include 1-18, 5-14, **PR**
I(nitialize 2-20
initialize disks 3-41, 7-53
input 2-7, 2-22, **PR**
INPUT 2-7, PR
I(nsert 4-8, 4-25
INSERT **PR**
INTERACTIVE **PR**
INTERFACE 1-21, **5-10**
Interpreter 1-11, **AG**
interrupts AG,
also see ATTACH **PR**
intrinsics **PR**
I/O errors B-1, AG
IORESULT **PR**, B-1

J

J(ump 4-13, **4-29**

K

KEYBOARD 3-5, PR, **E-1**
K(olumn **4-30**
K(runch **3-21**, 3-39

L

L(dir 3-23
LENGTH **PR**
library 1-20, **5-21**, AG
LIBRARY 1-20, **5-21**
library text file 1-20, 2-7, 2-9
L(ink 1-21, 2-4, 2-21, **5-18**,
Linker 1-21, 2-21, **5-18**,
AM, AG
list directory 3-23
log see M(onitor
long integers **PR**
lost files **3-42, 7-22, 7-41**

M

macro **AR**
M(ake **3-26**, 3-42
M(argin **4-31**, 4-42
MARK **PR**
MARKDUPDIR 3-45, **7-23**
markers see J(ump and S(et
MEMLOCK **PR**, AG
memory allocation and
management PR, AR, AG
MEMSWAP **PR**, AG
M(onitor 1-17, 2-22
MOVELEFT **PR**
MOVERIGHT **PR**
M(unch 4-31

N

Native Code Generator 7-45
N(ew 1-14, 3-6, **3-27**
NEW **PR**

O

Operating System 1-6, 2-6,
2-14, AG, also see System
output 1-17, 2-7, **PR**
OUTPUT 2-7, HB

P

PACK **PR**
packed variables PR, AG
P(age 4-13, **4-33**
Pascal **PR**, AM, 5-5, 6-3
PATCH 7-10
P-code 1-6, **AG**
POS **PR**

P-machine 1-6, AR, **AG**
P_MACHINE **AG**
prefix 1-15, 2-7, 3-28
P(refix 2-7, **3-28**
prefix disk 1-15, 2-7
Print Spooler 7-44
priority AG, also see
concurrent processes
PRINTER: 1-15, 3-3, PR
Printer Configuration 7-50
printer, serial 4-10
PROCEDURE see routine
program headings **PR**
PROCESS **PR**
PROCESSID see START
promptline 1-5, **2-3**
pseudo comments **PR**
PUT **PR**
PWROFTEN **PR**

R

READ **PR**
READLN **PR**
RECOVER 7-43
recovering lost files **3-42, 7-24, 7-41**
REDIRECT 2-13
redirection 2-10
RELEASE **PR**
R(emove 3-29
R(eplace 4-36
RESET **PR**
residence (in memory)
see memory allocation
REWRITE **PR**
routine 1-21, PR, AR
R(un 1-6, 1-14, 2-4, 2-24, PR
RS232SET 4-15

S

S(ave 1-14, 3-10, **3-31**
SCAN **PR**
Screen Control Unit 5-26
SEEK **PR**
segment 1-19, 5-5, PR, AG
segment routine 5-5
semaphores **PR, 6-6**
SEMINIT **PR**
separate compilation 1-21, 5-5, PR
serial printer 4-10
set **PR**
S(et 4-39
SETBAUD Utility 7-49
SETUP Utility 7-51
SIGNAL **PR**
SIZEOF **PR**
size specification (files) 3-26
special keys E-1
stack **AG**
START **PR, 6-3**
STR **PR**
strings **PR**
swapping 5-5, AG, and see
memory allocation
System 1-5, 1-7, 1-9, **2-3**, AG
SYSTEM.ASSMBLER 1-10,
2-17, **AR**, AG
SYSTEM.COMPILER 1-10,
2-18 PR, AG
SYSTEM.EDITOR 1-10,
2-19, AG
SYSTEM.FILER 1-10, 2-19,
AG
SYSTEM.LIBRARY 1-10,
1-20, **5-26**, AG
SYSTEM.LINKER 1-10,
2-21, 5-18, AG

SYSTEM.LST.TEXT AG, PR
SYSTEM.MISCINFO 1-11, 1-17, 2-21, **7-54**, AG
SYSTEM.PASCAL 1-10, 3-22, AG
SYSTEM.STARTUP 1-10, 2-20, AG
SYSTEM.SYNTAX 2-18, AG
SYSTEM.WRK.CODE 1-12, 2-17, 3-6, AG
SYSTEM.WRK.TEXT 1-12, 2-17, 3-6, 4-34, AG

T

Tape Utility 7-43
text **PR**
text editing see Editor
textfiles 1-13, **3-3**, 3-31, 4-6, PR
TIME **PR**
transcendental functions **PR**
T(ransfer 3-14, **3-33**
TRUNC **PR**
Turtle Graphics 5-40
TV Adjust Utility 7-48

U

U(ser restart 2-24
utility, RS232SET 4-15
UNIT 1-22, 5-7
unit numbers see device numbers
UNITBUSY **PR**
UNITCLEAR **PR**
UNITREAD **PR**
UNITSTATUS **PR**
UNITWAIT **PR**
UNITWRITE **PR**
UNPACK **PR**
untyped files **PR**
updating (a workfile) 4-34
USERLIB.TEXT 2-9, 5-7, 5-16
USES 5-16, PR
utilities 7-5

V

VARAVAIL **PR**
VARDISPOSE **PR**
VARNEW **PR**
V(erify 4-44
volume 1-14, 3-5, 3-13, 3-23
volume names 1-14, 3-7
volume numbers see device numbers
V(olumes 3-38

W

WAIT **PR**
W(hat 3-38
wildcards 3-10
workfile 1-11
WRITE **PR**
WRITELN **PR**

X

eX(amine 3-39
eX(change 4-44
eX(ecute 2-5, 2-7, 2-25
XREF 7-24

Z

Z(ap 4-46
Z(ero 3-40

Product Comment Form

Users' Guide 6936526

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

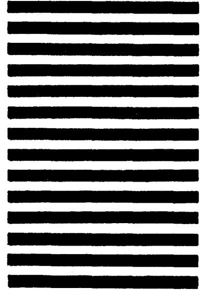
Name ______________________________

Address ______________________________

City ______________________ State ______________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328–W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6936526

Printed in United States of America