UCSD P-SYSTEM™

USERS' MANUAL SUPPLEMENT

VERSION IV

April 1982
SofTech Microsystems, Inc.
San Diego, California

ACKNOWLEDEGMENTS: This document was edited at SofTech Microsystems by Paula Johnson and Stan Stringfellow.

TABLE OF CONTENTS

I INTRODUCTION

This supplement describes new and upgraded portions of the Version IV UCSD p-System. It is an update to the UCSD p-System Users' Manual, Version IV.0 and the UCSD p-System Installation Guide.

This document is divided into four chapters. Chapter I (this introduction) gives an overview of what is covered in the rest of the supplement. Chapter II describes the new and upgraded standard facilities which are available with each p-System purchase. Chapter III covers features which may be optionally purchased with the p-System. Chapter IV contains information pertaining to the installation of adaptable systems with the current version IV p-System.

The following paragraphs summarize chapters II through IV and explain how the information in this supplement should be used in conjunction with the Users' Manual and the Installation Guide.

I.1 New and Upgraded Standard Facilities

The following topics are covered in Chapter II.

Symbolic Debugger

The P-code debugger has been upgraded to allow the specification of break point locations with a line number instead of a P-code offset. Also, variables can be specified by name rather than data offset. The information in the supplement replaces the information in the Users Manual, Version IV.0.

Event Handling

An event is a low level I/O action (e.g. an interrupt or the pressing of a key). The SBIOS can now notify the p-System that an event has occurred. If desired by the programmer, the occurrence of a given event will signal a Pascal semaphore. Use this part of the supplement as an update to the Users Manual, Version IV.0.

New File Handling Facilities

Subsidiary volumes is a file organization tool that creates a two-level file hierarchy on principal (or physical) disks. This mechanism is useful on large storage devices such as Winchester disk drives. Subsidiary volumes, can increase the number of files that may be stored on a disk.

User-defined serial devices is a new feature that allows the user to access more than the standard three serial devices.

The V(olumes command of the filer has been upgraded to display additional information that indicates the size, in blocks, of the disks that are on-line. Also, information pertaining to subsidiary volumes is displayed.

A new filer command called F(lip swap/lock allows memlocking of the filer's

code segments. This is a convenient mechanism that can be used if your system has enough memory. It allows the user to remove the disk containing the SYSTEM.FILER (and the boot disk, if they are different) while the filer is being used. Use this part of the supplement as an update to the Users Manual, Version IV.0.

The file management units allow programs to simulate the functions of the filer. For example, directories may be listed and information about files may be obtained. This part of the supplement is new information which is not found in the Users Manual, Version IV.0.

Error Message Handing

The programmer may now alter the way in which the p-System displays error messages. The location of the message on the screen and, in some cases, the actual message may be changed. The error message is then erased after the proper user input, and the cursor is returned to its previous position whenever possible. These features should be especially useful to applications developers. Section II.2 of the Users Manual, Version IV.0, is still correct except for the format of the error messages. There are additional error handling features described in Chapter IV of this supplement under SETUP. Execution errors are correct as described in Appendix A of the Users Manual, Version IV.0.

Compiler Enhancements

The Selective USES statement specifies to the compiler which identifiers from a unit are needed. The compiler then selects only the relevant identifiers thereby reducing compile-time symbol table space requirements. The selective USES statement is designed for compiling programs that use units with very large interface sections, only a small portion of which is required. The compiler enchancements section is an update to the Users Manual, Version IV.0 and the UCSD Pascal Handbook.

Extended Memory

It is now possible to place the code pool outside the stack/heap area within what is called "extended memory". The extended memory area may be as large as 64K bytes. This means that the p-System may run in configurations of up to 128K bytes with the current release. The extended memory section of the supplement is new information that is not found in the Users Manual, Version IV.0. See the section on SETUP in Chapter IV of this supplement to install this feature.

Echoed Characters

The user may now specify the characters that the keyboard will echo to the screen. Besides the standard ASCII character codes (0 through 127), the codes 128 through 255 may also be echoed if desired. This part of the supplement is new information that is not found in the Users Manual, Version IV.0. See the section on SETUP in Chapter IV of this supplement, to install this feature.

Editor Enhancements

This section describes user-defined tab stops, an addition to the S(et E(nvironment display, byte sex independence of text files created by the editor, and editor control characters. Use this part of the supplement as an update to the Users Manual, Version IV.0.

The 8086 and 68000 Assemblers

This section describes the 8086 and 68000 Assemblers. These assemblers are new to the p-System and are not described in the Users' Manual.

Turnkey Applications Facilities

The p-System will now recognize a user-written executable code file called SYSTEM.MENU. This file is executed automatically whenever the main system prompt line is normally displayed. This facility is useful for creating applications environments in which users will not be aware of the underlying p-System environment. Also, the p-System is now available as a turnkey package which does not include such system components as the filer and editor.

Performance Monitor

It is now possible to write a unit called PERFOPS and place it in the operating system. This unit allows the user to be cognizant of various p-System activities such as fault handling and segments being removed from memory.

I.2 New Optional Facilities

This section describes new optional facilities that are not found in the Users Manual, Version IV.0. The following topics are covered in Chapter III.

Native Code Generation

A new facility called the native code generator translates P-code, created by the p-System compilers, into native code, or the machine language of the host procesor. The resulting code file executes faster than an untranslated code file.

Print Spooling

The print spooler is a new facility that allows files to be sent to the printer during regular use of the p-System. You may, for example, print files and use the editor at the same time.

CP/M Xenofile

This facility allows you to access files on CP/M disks. This is accomplished by calling XenoFile from Pascal, BASIC, or FORTRAN just as the CP/M operating system routines are called.

Turtlegraphics

Turtlegraphics is a new package for creating and manipulating images on a graphics display. Routines are available to control the colors displayed, draw figures, alter old figures, save figures in disk files and retrieve them, etc.

I.3 Installation Guide Supplement

This section should be used in conjunction with the Installation Guide to install or upgrade adaptable systems. Even if you do not have an adaptable system, you should read through the topics below, or in the introduction to Chapter IV, to determine if any of the material presented is pertinent to your needs. Chapter IV discusses the following topics.

Adaptable System Disk Packaging

Adaptable systems are now shipped in a new format as described in this section.

Event Installation

To install the new event facility, two new SBIOS routines must first be written: QUIET and ENABLE. Then EVENT (a routine within the BIOS) may be called from the SBIOS. (The routines QUIET and ENABLE are required in any case in order to run the current version IV p-System.)

New Definition for PRNSTAT

The SBIOS routine PRNSTAT now accepts a new parameter as described in this section. PRNSTAT must be updated to run the current p-System.

Floating Point Number Configuration

This section describes how to configure the p-System for real numbers. No floating point package, two word floating point representation, or four word floating point representation may be used.

Installing GOTOXY

GOTOXY is slightly altered in the current release to allow for keeping track of the cursor on the screen.

p-System Support for ANSI Terminals

The p-System now can support American National Standards Institute (ANSI) terminals. If you have such a terminal you should read this section. This section should also be read by anyone who plans to Library or memlock Screenops.

The 8086 Adaptable System

This section describes the details of bringing up the adaptable system on the 8086 processor.

SETUP

The SETUP utility has been upgraded in accordance with the rest of this release. Echoed characters, extended memory, print spooling, subsidiary volumes, user-defined serial volumes, and code segment alignment are all affected by SETUP.

Installing Turtlegraphics

If you purchased Turtlegraphics as an adaptable package, you should read this section in order to install it.

Adaptable System Release Disk Directories

This section lists the directories of the adaptable system release disks. If you purchased the UCSD p-System as an adaptable system, reference these listings.

II NEW AND UPGRADED STANDARD FACILITIES

Chapter II details new and upgraded p-System standard features including the symbolic debugger, event handling, new filer facilities, error message handling, compiler enhancements, extended memory, echoed characters, editor enhancements, the 8086 and 68000 assemblers, file management units, system initialization enhancements, and performance monitor.

II.1 The Symbolic Debugger

This section of the Users' Manual Supplement, Version IV is a complete description of the P-code symbolic debugger. Use of the symbolic debugger is described in Section II.1.8.

The symbolic debugger is a tool for debugging compiled programs, which can be called from the main system prompt line or can be called during the execution of a program (when a breakpoint is encountered). Using the symbolic debugger, memory may be displayed and altered, P-code may be single-stepped, and markstack chains may be displayed and traversed.

There are no promptlines explaining the debugger commands because such prompts would detract from the information displayed by the debugger itself. However, when a command is entered, the system displays several short prompts that may ask for information.

Many of the debugger commands require two characters (such as "LP" for L(ist P(code, or "LR" for L(ist R(egister). To exit the program after typing the first character, press <space> to recall the main mode of the debugger.

To use the debugger effectively, a user must be familiar with the UCSD P-machine architecture and must understand the P-code operators, stack usage, variable and parameter allocation, etc. These topics are discussed in the Internal Architecture Guide.

A compiled listing of the program is a helpful debugging tool. It helps the user determine P-code offsets and similar information and should be current.

The debugger is a low-level tool, and as such, must be used with caution. If the debugger is used incorrectly, the p-System can fail.

II.1.1 Entering and Exiting the Debugger

Press D(ebugger to enter the debugger from the main system prompt line. If the debugger is entered in a fresh state, the system displays the following prompts.

```
DEBUG [version #]
(
```

A fresh state means that the debugger was not previously active, and no breakpoints are currently enabled. If the debugger is entered in a non-fresh state,

only the left parenthesis "(" appears.

Exit the debugger by pressing Q(uit, R(esume, or S(tep. The Q(uit option disables the debugger. If the debugger is re-invoked, it is in a fresh state. The R(esume option does not disable the debugger and execution continues from where it left off. The debugger is still active; and if it is re-invoked, it is in a non-fresh state. The S(tep option executes a single P-code and automatically re-invokes the debugger in a non-fresh state.

If a program is running under the debugger's R(esume command, it may force a return to the debugger by calling the HALT intrinsic (described in the Users' Manual). In fact, any run-time error causes a return to the debugger, if the debugger is active while the program is running.

The debugger may be memlocked or memswapped (see the descriptions of those intrinsics) by using the M(emory command at the outer level. "ML" memlocks and "MS" memswaps the debugger.

II.1.2 Using Breakpoints

To enter the debugger while a program is running, but not alter the program's code, use the debugger to set breakpoints. Press B(reakpoint and then use either the S(et, R(emove, or L(ist command. To set a breakpoint, press S(et after pressing B(reakpoint. There are, at most, five breakpoints numbered 0 through 4. The system displays four prompts asking for information. The first prompt is--

Set Break #?

Enter a digit in the range 0..4 and press <space>. The next prompt is--

Segname?

Enter the name of the desired segment and press <space>. The next prompt is--

Procname or #?

Enter the number of the desired procedure and press <space>. The final prompt is--

Offset #?

Enter the desired offset within the procedure and press <space>. The system sets a breakpoint, and if that segment, procedure, and offset are encountered during execution resumption, the debugger is automatically re-invoked.

Use a compiled listing of the program to determine the location of the breakpoint. If no compiled listing is available, use the text file viewing facility. See Section II.1.4.

To set a breakpoint that differs only slightly from the one most recently set, press <space> for the break number or segment. The system uses the previous breakpoint's information. For example, to break in the same segment and procedure but with a different offset, enter a space for everything except the offset.

To remove a breakpoint, press B(reakpoint, then press R(emove. The prompt,

Remove break #?

appears. To remove a breakpoint, enter its number followed by a <space>.

To list the current breakpoints, press B(reakpoint and then press L(ist.

II.1.3 Viewing and Altering Variables

The V(ar command allows the system to display data segment memory. It is another two-character command that must be followed by G(lobal, L(ocal, I(ntermediate, E(xtended, or P(rocedure. If G(lobal or L(ocal is selected, the system displays the following prompt.

Offset #?

Enter the desired offset into the data segment.

If I(ntermediate is selected, the system displays the following prompt.

Delta Lex Level?

Enter the appropriate delta lex level for the desired intermediate variable.
If E(xtended is selected, the system displays the following prompt.

Seg #? Offset #?

Enter the appropriate segment number and offset number for the desired extended variable.

If P(rocedure is selected, the system may display an offset within a specified procedure. The following prompts are displayed in sequence.

Segment name? Procname or #? Varname or Offset #?

When any of these options are used, the system displays a prompt similar to the following line.

```
( !) S=INIT P#1 VO#1 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa
```

This example is a portion of the local activation record for segment INIT, procedure 1, variable offset 1, at absolute hex location 2C1A. Following this, eight bytes are displayed, first in HEX and then in ASCII (a dash "-" indicates that the character is not a printable ASCII character).

To view surrounding portions of memory press V(ar. After a line has been displayed by the V(ar command, a "+" or "-" may be entered. This displays the succeeding or preceding eight bytes of memory.

The eight bytes that are currently displayed may be altered. If a "/" is typed, then the line may be altered in hex mode. If a "\" is typed, then the line may be altered in ASCII mode. When altering in hex mode, any characters that are to be left unchanged may be skipped by typing <space>. In the ASCII mode, any characters to be left unchanged may be skipped by typing <return>.

It is possible to change the frame of reference from which the global, local, and intermediate variables are viewed. This can be done by using the C(hain command. After "C" is typed the U(p, D(own and L(ist options are available. If "L" is typed, all of the currently existing mark stack control words are displayed, with the most recently created one first. An entry in the list resembles the following line.

```
(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA
msenv=FEE8
```

This corresponds to a mark stack control word with the indicated static link (msstat), dynamic link (msdyn), interpreter program counter (msipc), and erec pointer (msenv). The indicated segment (HEAPOPS), procedure (#3), and offset (#23) are the return point for the procedure call which created the MSCW.

If the U(p or D(own options are used, the frame of reference moves up or down one link and the frame of reference for variable listings (using the "V" command) changes accordingly.

II.1.4 Viewing Text Files from the Debugger

To view a text file from the debugger, press F(ile. The system displays the following prompt:

```
Filename?  First line #?  Last line #?
```

Enter the name of the text file to be viewed followed by <space>. The .TEXT portion of the file name is optional. Then enter the first and last line numbers that delimit the portion of text that the user wishes to view. This command lists as many lines as possible in the window from "first line" to "last line" of the indicated file.

The F(ile command is useful for debugging (especially using symbolic debugging) when a hard copy of the relevant compiled listing is not available. Using this command, the user can view source files on disk and disk files containing compiled listings without leaving the debugger.

II.1.5 Displaying Useful Information

Whenever control is returned to the debugger (e.g. after a single step operation, or when a breakpoint is encountered), it displays various information if it is desired. This information may include P-machine registers, the current P-code operator, the information in the current markstack, or any specified memory location. In order to select what will be displayed, the E(nable mode should be used. After typing 'E', the following options are available at the command level, R(egister, P(code, M(arkstack, A(ddress, and E(very (all of the preceding). Any or all of these options may be enabled at the same time.

If R(egister is enabled, a line is displayed after each single step. The following line is an example of that display.

```
(rg) mp=F082 sp=F09C erec=FEE8 seg=9782 ipc=01C3 tib=0493 rdyq=2EBC
```

If P(code is enabled, a line such as the following is displayed after each step:

```
(cd) S=HEAPOPS P#3 O#23  LLA 1
```

If M(arkstack is enabled, a line such as the following is displayed after each step:

```
(ms) S=HEAPOPS P#3 O#23 msstat=347C msdyn=F0A0 msipc=01DA
msenv=FEE8
```

If A(ddress is enabled, the system generates a display like the following line.

```
(a ) S=HEAPOPS P#3 O#23 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa
```

To initialize this address to a given value, use A(ddress mode at the outer level. Press A(dress and the system displays the following prompt.

```
Address ?
```

Enter the absolute address in hex. The system displays eight bytes starting at that address. Also, that address is now displayed if the E(nable A(ddress option is on.

Enabling E(very causes all of the above options to be enabled.

The D(isable mode disables any of the options just described. The L(ist mode lists any of the above options.

II.1.6 Disassembling P-code

At the debugger's outer level, there is a P-code option that displays the P-code mnemonics for selected portions of code. This option asks for:

```
Segname?
Procname or #?
Start Offset #? and End Offset #?
```

The indicated portion of code is then disassembled. This may be useful during single-step mode if you wish to look ahead in the P-code stream. This mode may be exited before it reaches the ending offset by typing <break>; control returns to the debugger.

II.1.7 Example of Debugger Usage

Suppose the following program is to be debugged:

```
Pascal Compiler IV.0

1 0 0:d 1  {$L LIST.TEXT}
2 2 1:d 1  PROGRAM NOT_DEBUGGED;
3 2 1:d 1  VAR I,J,K:INTEGER;
4 2 1:d 4      B1,B2:BOOLEAN;
5 2 1:0 0  BEGIN
6 2 1:1 0    I:=1;
7 2 1:1 3    J:=1;
8 2 1:1 6    IF K <> 1 THEN WRITELN ('Whats wrong?');
9 2  :0 0  END.

End of Compilation.
```

First we enter the debugger and set a breakpoint at the beginning of the IF statement:

```
(BS) Set break #? 0 Segname? NOTDEBUG Procname or #? 1 Offset #? 6
(EP)
(R)
```

After setting the breakpoint we enable P-code (EP) and resume (R). Now we execute the program above, and when it reaches offset 6, the debugger is entered. We single-step twice:

```
Hit break #0 at S=NOTDEBUG P#1 O#6
 (cd)  S=NOTDEBUG   P#1 O#6   SLDO1
 (cd)  S=NOTDEBUG   P#1 O#7   SLDC1
 (cd)  S=NOTDEBUG   P#1 O#8   NEQUI
```

We see that our first single-step did a short load global 1. (Note: This put K on the stack. K is NOT global 3; I is global 3, J is global 2, and K is global 1. Every string of variables (such as 'I, J, K' in a declaration) is allocated in reverse order. Boolean B1, which follows, is at offset 5, and B2 is at offset 4. Parameters, on the other hand, ARE allocated in the order in which they appear.) The second single-step did a short load constant 1 onto the stack. Now we are about to do an integer comparison (<>). But this is where our error shows up, so we decide to look at what is on the Stack before doing this comparison:

```
(LR)
(rg)  mp=EB62   sp=EB82   erec= ...
(A )  Address? EB82
(a )                EB82: 01 00 C5 14 ...
```

We list the registers and then look at the memory address to which register sp points. We discover a 1 on top of the stack (01 00: this is a least-significant-byte-first machine) followed by a word of what appears to be garbage. This leads us to suspect that K was not initialized. Looking over the listing, we quickly realize that this is the case.

II.1.8 Symbolic Debugging

The symbolic debugging feature allows specification of variables by name, rather than P-code offset. Also, breakpoints and portions of code to be disassembled may be indicated by procedure name and line number, rather than procedure number and P-code offset.

Having a current compiled listing of the code in question is still essential for serious debugging efforts.

To use symbolic debugging, it is necessary that the code being debugged is compiled with the $D+ option (the Users Manual, Version IV.0 describes compiler options in general). The $D+ option, which defaults to $D-, instructs the compiler to output symbolic debugger information for those portions of a program that are compiled with $D+ turned on. Once a program is debugged, it should be recompiled without symbolic debugger information, because this information increases the size of the code file.

Using symbolic debugging, breakpoints may be specified by procedure name and line number for all statements covered by the $D+ option. The B(reakpoint command will request:

"Procname or #?"

Enter the first eight characters of the procedure name. The next line displayed will be--

"First#__ Last#__ Line#?"

The underlines will actually be values that define the range of line numbers available to you within the specified procedure. (These line numbers appear on compiled listings.) You should then enter the desired line number for your breakpoint.

Variables within a given routine may be specified by name (rather than data segment offset number) if at least one statement within that routine is compiled by $D+. The V(ar command allows specification of G(lobal, L(ocal, I(ntermediate, or P(rocedure variables in this manner. E(xtended variables are not allowed to be specified symbolically. The V(ar command will prompt:

"Varname or Offset #?"

You may enter the first eight characters of the declared identifier. A line similar to the following will then appear:

(1) S=INIT P=FILLTABL V=TABLE1 2C1A: 0B 05 53 43 41 4C 43 61 --SCALCa

The segment is INIT, the procedure is FILL_TABLES, and the variable is TABLE1.

Similarly, the code to be disassembled by the P-code command can be specified symbolically for all portions of code covered by the $D+ option. This command will request:

Procname or #?

Enter the first eight characters of the procedure name. You will then be prompted as follows:

```
First #__ Last #__ Start Line#?  End Line#?
```

The underlines will actually be the boundaries that are available to you. You should enter the desired starting and ending line numbers. The specified code will then be disassembled.

II.1.9 Symbolic Debugging Example

To use symbolic debugging, some part of a Pascal compilation unit must be compiled with the {$D+} compile-time directive. After this code has been generated, it is possible to reference variables and procedures by name rather than offset. The following example is a small Pascal program that has been compiled with the 'D' option.

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

Pascal Compiler IV.1 c5s-4      3/ 4/82      Page   1

=====================================================

    1   0     0:d     1       {$D+}
    2   2     1:d     1        program example;
    3   2     1:d     1        var a,b,c:integer;
    4   2     1:d     4
    5   2     1:d     4          procedure set_c_if_d;
    6   2     2:d     1          var d:boolean;
    7   2     2:0     0          begin
    8   2     2:1     0            d:=a>b;
    9   2     2:1     5            if d then
   10   2     2:2     8              c:=a*b;
   11   2     1:0     0          end;
   12   2     1:0     0
   13   2     1:0     0        begin
   14   2     1:1     0          a:=0;
   15   2     1:1     3          b:=5;
   16   2     1:1     6          set_c_if_d;
   17   2      :0     0        end.

End of Compilation.
```

The following listing is an example of a debug session.

```
Debug [x15]
(BS) Segname=EXAMPLE Procname or # = SETCIFD
       symbolic seg not in mem Line#? 8
(R )

Hit break#0 at S=EXAMPLE P=SETCIFD L#8
(BS) Segname=EXAMPLE Procname or # = SETCIFD
       First#8 Last#10 Line#? 9
(R )

Hit break#1 at S=EXAMPLE P=SETCIFD L#9
(VL) Varname or offset#? D
(I ) S=EXAMPLE P=SETCIFD V=D  E7B2 : 0000 9448 BEE7 190C-H-
(Q )
```

The first time the debugger is entered, the program example is not in memory and hence the symbolic segment is not in memory. However, a breakpoint can still be set symbolically providing the user knows on which line number to stop. For the second breakpoint, the symbolic segment is in memory; because of this, its first and last line numbers are given.

Notice the variable 'D' was accessed symbolically, and its contents are displayed.

If the user tries to access symbolically when the actual code segment is in memory and its symbolic segment counterpart is not present, the system will display the error message 'symbolic seg not in mem'. Use the 'Z' command in the symbolic debugger to find out if symbolic information is available for a particular segment.

The 'Z' command lists all of the principal segments with their environment list. For example, the following example is a partial list of the principal segments (EDITOR and EXAMPLE) and their environments. The lowercase name 'example' is the symbolic segment for 'EXAMPLE'. The existence of 'example' indicates that symbolic debugging information is available for at least one procedure in 'EXAMPLE'.

```
(Z ) the sib is EDITOR
  1 KERNEL
  2 EDITOR
  3 INITIALI
  4 PASCALIO
  5 EXTRAIO
  6 GOTOXY
  7 STRINGOP
  8 EXTRAHEAP
  9 FILEOPS
  10 OUT
  11 COPYFILE
  12 ENVIRONM
  13 PUTSYNTA
  14 EDITCOR

the sib is EXAMPLE
  1 KERNEL
  2 EXAMPLE
  3 example
```

II.1.10 Interacting with the Performance Monitor

The "I" command will call the PM_Interactive procedure within the operating system if the performance monitor is enabled. The user may, in this way, gain access to fault handling data, and information concerning what programs have started and completed execution. For more information, see Section II.11 entitled Performance Monitor.

II.1.11 Summary of the Commands

A(ddress	Displays a given address
B(reak point	Segment, procedure and offset must be specified
S(et	Allows a break point (0 through 4) to be set
R(emove	Allows a break point to be removed
L(ist	Lists current break points
C(hain	Changes frame of reference for V(ariable command
U(p	Chains up mark stack links
D(own	Chains down mark stack links
L(ist	Lists current mark stacks
F(ile	Allows viewing of text files
E(nable	Enables the following to be displayed
D(isable	Disables the following from being displayed
L(ist	Lists the following
R(egister	The registers: mp, sp, erec, seg, ipc, tib, rdyq
P(code	Current P-code mnemonic
M(arkstack	Mark stack display
A(ddress	A given address
E(very	All of the above
I(nteractive	Interacts with the performace monitor
M(emory	
L(ock	Memlocks the debugger
S(wap	Memswaps the debugger
P(code	Dissassembles a given procedure
Q(uit	Quits the debugger, 'fresh' state if re-entered
R(esume	Exits debugger, debugger remains active, 'non-fresh'
S(tep	Single steps P-code and returns to debugger

V(ariable

G(lobal	Displays global memory
L(ocal	Displays local memory
I(nter	Displays intermediate memory
P(roc	Displays data segement of given procedure
E(xtended	Displays variables in another segment
Z(	Displays segment lists.

II.2 Event Handling

Event handling is a new feature which allows Pascal programs to respond to low level events. These events include such things as hardware interrupts and low level I/O actions.

The new print spooler, for example, is able to function concurrently with other p-System activity through this mechanism. Every time a character is entered at the keyboard, an event occurs (on those systems set up for print spooling). The software can take advantage of this knowledge and switch back and forth between an editing session, for example, and the print spooling process.

To install the event handling facility, the user must add two relatively simple routines to the SBIOS. These routines are called QUIET and ENABLE and are used by the p-System to control event handling. It is also necessary to augment the SBIOS with calls to the new BIOS routine EVENT. These calls will inform the upper levels of the p-System that it has recognized the event with that number. This is discussed in Chapter IV of this supplement.

The remainder of this section describes how to use the event handling facility after it has been installed.

In order to understand event handling, it is necessary to understand Pascal processes. A process is a routine similar to a function or procedure. Processes may run concurrently with each other and with the main program. Once a process is STARTed, it may WAIT on a semaphore before continuing execution. While a process waits, other processes (including the main program) may resume execution. When the semaphore in question is SIGNALed, the waiting process may have the opportunity to resume execution. (For more information concerning processes, see The UCSD Pascal Handbook.)

An event is assigned a particular number. Whenever that event occurs, the SBIOS informs the p-System that it has recognized the event with that number. System events use the numbers 0..31 and user-defined events may use the numbers 32..63.)

The Pascal programmer should use the ATTACH intrinsic (see the Users Manual, Version IV.0 and The UCSD Pascal Handbook) to assign event numbers to semaphores. When an event occurs, the p-System SIGNALs the corresponding semaphore. If no semaphore is ATTACHed to the event number, no action is taken.

Therefore, the occurrence of a low level event may possibly cause a particular Pascal process to receive processor time.

The following example shows how events might be used. This program assumes that the SBIOS has been set up to cause event 32 whenever input is received from a special user device. It executes normally until the user device input is noticed, at which point it handles that input and returns to normal execution.

```
PROGRAM USER_EVENT;
CONST
  EVENT_NUM = 32;
VAR
  PID: PROCESSID;
  S: SEMAPHORE;
  FINISHED: BOOLEAN;
        .
        .
        .
PROCESS HANDLE_USER_INPUT;
BEGIN
  WHILE TRUE DO
    BEGIN
      WAIT(S);
      IF FINISHED THEN EXIT(HANDLE_USER_INPUT);
         .
         .
         .
    END
END;

BEGIN
  FINISHED:=FALSE;
  SEMINIT(S,0);
  ATTACH(S,EVENT_NUM);
  START(HANDLE_USER_INPUT,PID,300,200);
       .
       .
       .
  FINISHED:=TRUE;
  SIGNAL(S);
END.
```

In the preceding example, the semaphore S is ATTACHed to event number 32. Then process HANDLE_USER_INPUT is STARTed with a high priority of 200. HANDLE_USER_INPUT then WAITs on S while the rest of the program continues normal execution. Whenever the SBIOS notices input from the user device, event 32 is caused. This event SIGNALs S, and because HANDLE_USER_INPUT has a high priority, it immediately receives processor time to take care of the input.

Each time HANDLE_USER_INPUT returns to the beginning of the loop, it WAITs on S and the main program resumes. Finally, the boolean FINISHED is set to true by the main program and S is SIGNALed directly. This causes an exit from the loop within HANDLE_USER_INPUT, and the program terminates.

NOTE: An occurrence of ATTACH using a given event number will override any previous occurrence of ATTACH with that event number. This means that the first semaphore will no longer be ATTACHed to the event.

NOTE: To detach a previously attached semaphore from an event number, attach NIL to that event number. For example, Attach(NIL,EVENT_NUM); detaches S from EVENT_NUM in the preceding program example.

II.3 New File Handling Facilities

In previous versions of the UCSD p-System, there were 6 blocked devices (4, 5, 9, 10, 11, 12). Each blocked device could contain a maximum of 16 megabytes, and the p-System could access 96 megabytes of storage. Now the user can change the number of blocked devices, allowing the p-System to access more than 1.7 billion bytes of on-line storage. See Chapter IV under SETUP to allocate blocked device numbers.

Section II.3.1 covers subsidiary volumes, which allows the user to create subdirectories on a blocked device. Subsidiary volumes are especially useful for large storage devices such as Winchester Drives. See Chapter IV under SETUP to allocate device numbers for subsidiary volumes.

Section II.3.2 describes user-defined serial devices, which allow the user to access more serial devices.

Section II.3.3 describes the upgraded V(olumes command. This filer command now displays additional information pertaining to the size of disks. Also, information related to subsidiary volumes is displayed.

Section II.3.4 covers the new F(lip swap/lock filer command. This command allows memlocking of the filer's code segments on systems that have enough memory to do so. When the filer is memlocked, the disk containing SYSTEM.FILER (and also the boot disk, if it is different) may be removed during filer use. This facilitates the use of the filer, especially on two-drive systems.

Section II.3.5 documents the use of the filer P(refix command.

Section II.3.6 presents a new facility called file management units, which are used by a Pascal program to accomplish tasks normally performed by the filer.

II.3.1 Subsidiary Volumes

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

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

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

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

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

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

II.3.1.1 Creating and Accessing Subsidiary Volumes

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

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

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

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

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

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

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

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

```
Zero subsidiary volume directory?
```

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

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

When a subsidiary volume is created, it is automatically mounted and may be accessed and used like any other p-System volume. The filer command V(olumes will indicate that the new volume is on-line, and will show its corresponding device number (for example, #13:). Either the volume ID or the device number may be used when referencing the subsidiary volume. Files may now be placed on the new subsidiary volume, and all of the applicable file commands may reference it.

II.3.1.2 Mounting and Dismounting Subsidiary Volumes

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

A mounted subsidiary volume means that the p-System is aware of the existence of the volume and sets aside a device number for it (e.g., #13:). It is required that a subsidiary volume be mounted before it can be used. While it is mounted, only that specific subsidiary volume will correspond to that device number. A subsidiary volume stays mounted until it is dismounted. Once mounted, it is on-line anytime its principal volume is in the disk drive. It is off-line when the principal volume has been removed from the disk drive.

CAUTION: There is a danger of confusing the p-System if two principal volumes each contain a subsidiary volume in the same location with the same name. This might easily be the case where backup disks are used. If these principal volumes are swapped in and out of the same drive, and the similar subsidiary volumes are accessed, the filer may become confused in the same way that it can when any two on-line volumes have the same name.

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

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

The booting or initializing process will mount as many subsidiary volumes as it finds as long as there is room in the p-System unit table. (see Chapter IV under SETUP for the maximum number of subsidiary volumes.) If the unit table becomes full, no more subsidiary volumes will be mounted, and no warning is given.

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

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

```
Subsidiary Volume: M(ount, D(ismount
```

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

```
Mount what vol ?
```

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

```
Dismount what vol ?
```

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

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

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

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

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

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

```
Dismount what vol ? #14:
Dismount what vol ? VOL1:<return>
Dismount what vol ? VOL1:, DOC1:, -FUN-:<return>
```

The first example dismounts the subsidiary volume associated with the device number 14. The second example dismounts VOL1:, and the third example dismounts three subsidiary volumes.

II.3.1.3 Installation Information

It is very simple to install the subsidiary volume facility into a Version IV system if the user employs SETUP (described in Chapter IV of this supplement) to set MAX NUMBER OF SUBSIDIARY VOLS to the smallest convenient value. This will be the maximum number of subsidiary volumes that will be allowed to be mounted at one time. (Each additional subsidiary volume requires a few extra bytes within the p-System's unit table.) When this has been done, the subsidiary volume facility will be available.

II.3.2 User-Defined Serial Devices

The user-defined serial device facility is a new feature that allows the user to take advantage of serial I/O hardware capabilities. This facility, along with the standard serial I/O capabilities (CONSOLE:, REMIN:, and REMOUT:) can be used on machines with special serial I/O hardware.

The user may have up to 16 devices in addition to a print, console and a remote. User-defined serial devices may include additional printer ports, additional consoles, communication lines between users in a multi-user environment, etc.

This feature will not be available through the adaptable system BIOSes. See Chapter IV of this supplement under SETUP to configure the system to omit or to accommodate user-defined serial devices.

II.3.3 The Flip Swap/Lock Command

The following command is a new item on the filer prompt line.

F(lip swap/lock

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

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

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

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

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

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

II.3.4 The V(olumes Command

Some additional information has been added to the V(olumes display. The sizes of on-line disk volumes are now given. In the following example, three volumes are on-line. Typing "V" from the main filer prompt causes the display:

```
Vols on-line:
   1    CONSOLE:
   2    SYSTERM:
   4 #  WNCHSTR: [12000]
   5 #  FLOPPY1: [  494]
   6    PRINTER:
  12 #  FLOPPY2: [  500]
Root vol is - WNCHSTR:
Prefix is   - FLOPPY2:
```

After each disk volume, the number of 512-byte blocks that it contains is given in square brackets. This can be useful if the system uses disks of varying storage capacities. In the preceding example, the Winchester Disk on-line in drive #4: contains 12000 blocks of storage capacity, and the floppies on-line in drives #5: and #12: contain 494 and 500 blocks respectively.

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

```
Vols on-line:
  1    CONSOLE:
  2    SYSTERM:
  4 #  WNCHSTR: [12000]
  5 #  FLOPPY1: [  494]
  6    PRINTER:
 12 #  FLOPPY2: [  500]
 13 #  DOCS:    [ 3000] on volume WNCHSTR: starting at block 400
 14 #  PROGRMS: [ 3000] on volume WNCHSTR: starting at block 3700
 15 #  --FUN--: [ 3000] on volume WNCHSTR: starting at block 7040
Root vol is - WNCHSTR:
Prefix is   - FLOPPY2:
```

In this example, three subsidiary volumes on WNCHSTR: are mounted. They use device numbers #13:, #14:, and #15:. Each of these volumes contain 3000 blocks.

II.3.5 Filer P(refix Notes

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

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

II.3.6 File Management Units

The file management units may be used by Pascal programs to perform several of the tasks normally done by the filer. There are four units:

DIR.INFO.CODE
WILD.CODE
FILE.INFO.CODE
SYS.INFO.CODE

DIR.INFO provides directory information and may be used to list directories; parse file names into volume ID, file name, file type, and size specifications; change file names; change the date associated with a file or volume; remove files; grant exclusive access rights to a directory by task; release those access rights.

WILD provides wild card string matching facilities.

FILE.INFO allows the programmer to determine if files are opened, find the length of a file, determine what device contains a given file, find the volume ID that contains a file, extract the file title (with suffix) from a file name, find the starting block of a file, determine if a file is blocked or not, and return a file date.

SYS.INFO allows the user to determine the volume name and title of the work code file, the file title of the work text file, the unit number of the device containing the system (*) volume, and the volume name of the system (*) volume. SYS.INFO allows the user to determine and change the system (*) volume and the system date.

II.3.6.1 Directory Information (DIR.INFO)

Many applications require programs to access and modify directory information. This unit makes most directory operations easy to perform. There are other ways in which this might be done. The most common solution is to construct user programs that directly access the operating system's data structures. A better solution is to construct UCSD p-System Units that provide routines for accessing directory information in a controlled manner. The interfaces provided by well-designed units make directory information access much safer and easier.

This section describes a directory information unit (named DIR_INFO) which provides user programs with access to file system information.

The directory information unit provides the following capabilities:

Directory Information Access. For any on-line disk unit, DIR_INFO returns the volume name, volume date, number of disk files on volume, amount of unused space, and attributes of individual disk files.

Directory Manipulation. DIR_INFO provides routines for changing the date or name of a disk file or volume, removing files from a volume, and taking volumes off-line.

Wild Cards. DIR_INFO uses the UNIT WILD, which provides a wild card convention for pattern matching of string variables. Most DIR_INFO routines recognize the wild card convention in their file name arguments.

Error Handling. DIR_INFO defines a standard error result (similar to UCSD p-System I/O results) for routines involved with file names and directory searches.

Multi-tasking Support. DIR_INFO provides routines for protecting file system information from contention between concurrent tasks. These routines ensure that only one task can modify file system information at a time.

II.3.6.1.1 Notation and Terminology

A variant of Extended Backus-Naur Form (EBNF) is used as a notation for describing the form of wild cards and file names. Meta-words are words that represent a class of words; they are shown in the text by the use of angle brackets < >. Thus, the words trout, salmon, and tuna are acceptable substitutions for the meta-word <fish>. The following expression is an example of the the substitution.

<fish> = trout | salmon | tuna

The equal sign indicates that the meta-word on the left side can be substituted with the word on the right side. The bar (|) separates possible choices for substitution. The preceding example indicates that trout, salmon, or tuna can be substituted for <fish>.

An item enclosed in square brackets [ ] may be substituted into a textual expression. For example, [micro]computer can represent the text strings computer and microcomputer.

An item enclosed in braces { } can be substituted zero or more times into a textual expression. The following expression represents responses to jokes possessing varying degrees of humor.

<joke-response> = {ha}

Literal occurrences of characters or strings of characters are delimited by quotes to avoid confusing them with notational definitions

For example: left-bracket = "<" / "{" / "["

The terminology used in describing the file system is defined in the Users Manual, Version IV.0. The term <file-object> is used throughout this document; it is a generic term encompassing serial and disk volumes, disk files, and unused areas of disk volumes.

II.3.6.1.2 Wild Cards

Most DIR_INFO routines allow <wild cards> in their file name arguments. A description of wild card facilities and operation is given in the WILD UNIT definition.

II.3.6.1.3 File Name Arguments

Most DIR_INFO routines accept file name arguments. The file name specifies the volume and/or file to be accessed by the routine. See the Users Manual, Version IV.0 for a complete description of UCSD p-System files and file names.

File name syntax:

```
<file-name>      =  [volume-ID]  [file-ID]

<volume-ID>      =  [ "#" unit-number ":"|
                              volume-name":"|
                              ":" | "*" | "*:" ]

<file-ID>        =  [title suffix specifier]

<specifier>      =  "["number"]" | "["*"]"
```

Volume names and file titles may contain wild cards. Unit numbers and colons separating volume IDs and file IDs must appear literally; they must be independent of any wild card.

All DIR_INFO routines except D_Scan_Title ignore file length specifiers. File name conventions in DIR_INFO differ slightly from UCSD p-System file name conventions in some cases:

DIR_INFO considers an empty file name argument to specify the prefix volume; i.e., <file-ID> is empty (implying a volume reference), and <volume-ID> is empty (implying the prefixed volume). An empty string is not a valid file name in the p-System.

DIR_INFO interprets wild card file names of the form <vol-name>:= to be valid volume specifiers. This is consistent with DIR_INFO's definition of the "=" wild card, but inconsistent with the UCSD p-System Filer's interpretation of the "=" wild card; the filer does not accept file names of this form as volume specifiers.

II.3.6.1.4 File Type Selection

Some DIR_INFO routines accept a <file-type> parameter (named D_SELECT) which is used to specify the file objects to be accessed. (File objects include volumes, unused areas on disk volumes, temporary files, text files, code files, and data files.) The file type parameter is necessary because file names are not sufficient to completely specify all types of file objects (e.g. unused disk areas). The routines which generate directory information use both the file name argument and the D_SELECT parameter to determine the file objects on which to return information.

DIR_INFO defines a scalar type, which is used to specify file objects. D_SELECT is declared as a set of this type; a file object is selected by including its corresponding scalar in D_SELECT.

File object types:

```
D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_SVol, D_Temp, D_Free);

D_Choice   = Set Of D_NameType;
```

The scalar values are defined as follows:

D_Vol--Select all volumes matching the file name argument. Note that while volume names may contain wild cards, unit numbers must be specified literally.

D_Free--Select all unused areas of disk space on the volumes matching the file name argument.

D_Temp--Select all temporary files matching the file name argument. Files are considered temporary if they have been opened (but not yet locked) by a program.

D_Text--Select all text files matching the file name argument.

D_Code--Select all code files matching the file name argument.

D_Data--Select all data files matching the file name argument.

D_SVol--Select all svol files matching the file name argument.

II.3.6.1.5 File Dates

Disk files and disk volumes are assigned <file-dates>. File dates are stored in records of type D_Date_Rec and are accessed and modified by the DIR_INFO routines D_Dir_List and D_Change_Date.

D_Date_Rec is declared as follows:

```
D_DateRec = Packed Record
                    Month : 0..12;
                    Day   : 0..31;
                    Year  : 0..100;
                    End;{ D_DateRec }
```

A year value of 100 in a file date record indicates that the object is a temporary disk file. (This is a UCSD p-System file system convention.)

II.3.6.1.6 Error Results

All DIR_INFO routines which access file system information return a value reflecting the result of the file system operation. This result indicates either that the routine finished without errors or that an error occurred; valid information is not returned when routines return a result value indicating the occurrence of an error.
Error causing conditions include:

The specified files, volumes, or unused spaces can not be found in the disk directory.

The specified unit is off-line.

The file name argument has improper syntax.

The specified file name conflicts with an existing file.

In no cases can an error cause abnormal termination of a function; errors which cannot be identified explicitly by the routine are flagged by returning a result indicating that an unknown error has occurred.

DIR_INFO defines a scalar type to describe the possible errors encountered:

```
Type  D_Result=      (D_Okay,
                     D_Not_Found,
                     D_Exists,
                     D_Name_Error,
                     D_Off_Line,
                     D_Other);
```

Details on error results and the status of the returned directory information in the presence of an error can be found in the descriptions of each of the unit procedures.

II.3.6.1.7 DIR_INFO Interface

```
Unit DIR_INFO;

Interface
  uses
    (*$UWILD.CODE*)
    wild;

  Type
    D_DateRec = Packed Record
                  Month : 0..12;
                  Day   : 0..31;
                  Year  : 0..100;
                End;
    D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_SVol,
      D_Temp, D_Free);
    D_Choice = Set of D_NameType;
    D_ListP = ^D_List;
    D_List = Record
               D_Unit : Integer;
               D_Volume : String[7];
               D_VPat : D_PatRecP;
               D_NextEntry : D_ListP;
               Case D_IsBlkd : Boolean Of
                 True : (D_Start,
                         D_Length : Integer;
                         Case D_Kind : D_NameType Of
                           D_Vol,
                           D_Temp,
                           D_Code,
                           D_Text,
                           D_Data,
                           D_SVol:
                             (D_Title : String[15];
                              D_FPat  : D_PatRecP;
                              D_Date  : D_DateRec;
                              Case D_NameType of
                                D_Vol:(D_NumFiles:Integer)));
             End;
```

```
    D_Result = (D_Okay,
                D_Not_Found,
                D_Exists,
                D_Name_Error,
                D_Off_Line,
                D_Other);

Function D_Dir_List(D_Name : String;
                    D_Select : D_Choice;
                    Var D_Ptr : D_ListP;
                    D_PInfo : Boolean): D_Result;

Function D_Scan_Title(D_Name : String;
                    Var D_VolID,D_TitleID : String;
                    Var D_Type : D_NameType;
                    Var D_Segs : Integer)
                    : D_Result;

Function D_Change_Name(D_OldName,
                    D_NewName : String;
                    D_RemOld : Boolean)
                    : D_Result;

Function D_Change_Date(D_Name : String;
                    D_NewDate : D_DateRec;
                  D_Select : D_Choice) : D_Result;

Function D_Rem_Files (D_Name : String;
                    D_Select : D_Choice) : D_Result;

Procedure D_Lock;
Procedure D_Release;

Function D_Krunch (D_Unit:integer; D_Block:integer): D_Result;

Function D_Mount (D_File_Name:String): D_Result;
Function D_DisMount (D_Vol_Name:String): D_Result;
```

II.3.6.1.8 Detailed Description of Functions

Function D_Krunch (D_Unit:integer; D_Block:integer) D_Result;

This function will move the files on the volume specified by D_Unit. The block indicated by D_Block will be included in the resulting unused disk space area. Files located before D_Block will be moved forward (toward the directory) and files after it will be moved backward (toward the last track).
NOTE: Using D_Krunch on a volume that contains an executing or open file (including the operating system) may destroy the files. If Function D_Krunch changes the location of an open or executing file, the system will return data to the previous, not present location of the file.

Function D_Mount (D_File_Name:String):D_Result;

The D_File_Name parameter identifies a subsidiary volume that is to be mounted. Wild cards may be used. The volume will be mounted unless D_Result indicates otherwise.

Function D_DisMount (D_Vol_Name:String):D_Result;

The subsidiary volume identified by the D_Vol_Name parameter is dismounted. This volume must be a subsidiary volume.

**Function D_Scan_Title(D_NAME:String; Var D_VOLUME,
D_TITLE: String; Var D_TYPE: D_NameType; Var D_SEGS: Integer):D_Result;**

D_Scan_Title parses the UCSD p-System file name passed in D_NAME, and returns the file name's volume ID, file title, file type, and file length specifier. The function result indicates the validity of the file name argument. D_Scan Title does not determine whether or not D_Name actually exists.

Parameters and Function Results:

D_Scan_Title accepts the following parameters.

D_NAME--A string containing a UCSD p-System file name.

D_VOLUME--A string which returns the volume ID contained in D_NAME. If D_NAME contains no volume ID or if the volume ID is ':', D_VOLUME is assigned the system's default volume name. If the volume ID is '*' or '*:', D_VOLUME is assigned the system's boot volume name. Volume names assigned to D_VOLUME contain only upper case characters, and do not contain blank characters.

D_TITLE--A string which returns the file title contained in D_NAME. If D_NAME does not contain a file title, D_TITLE is assigned the empty string. File titles assigned to D_TITLE contain only upper case characters, and do not contain blank characters.

D_TYPE--A scalar which returns a value indicating the file type of the file name contained in D_NAME.

Definition of D_TYPE's scalar type:

```
D_NameType = (D_Vol, D_Code, D_Text, D_SVol, D_Data, D_Temp, D_Free,);
```

D_TYPE is set to D_Vol if the file title in D_NAME is empty. D_TYPE is set to D_Code if the file title is terminated by ".CODE" or to D_Text if the file title is terminated by ".TEXT" or ".BACK". D_TYPE is set to D_SVOL if the file title ends with .SVOL (a subsidiary volume). If none of the above holds true, D_TYPE is set to D_Data. Only the suffix of a file is used for determine what type it is. For example, the file name SYSTEM.COMPILER is returned as a data file because it's suffix is not .CODE .

D_SEGS--An integer that is assigned a value indicating the presence of a file length specifier in D_NAME. The value returned in D_SEGS is assigned as follows:

LENGTH SPECIFIER	D_SEGS VALUE
[<number>]	<number>
[*]	-1
<not present>	0

D_Scan_Title returns a function result of type D_Result. The only scalar values returned by D_Scan_Title are D_Okay and D_Name_Error; they have the following meanings:

D_Okay--No Error. All information returned by D_Scan_Title is valid.

D_Name_Error--Illegal file name syntax in D_NAME. The information returned by D_Scan_Title is invalid.

Programming Example:

```
Program Scan_Test;
Uses
   (*$UWILD.CODE*)
   wild,
   (*$UDIR.INFO.CODE*)
   DirInfo;
Var
   Name,
   Volume,
   Title      : String;
   Typ        : D_NameType;
   Seg_Flag   : Integer;
   Result     : D_Result;
   Ch         : Char;
   Begin { Scan_Test }
   Writeln('--D_ScanTitle Test');
   Repeat
      Writeln;
      Write('File name to parse: ');
      Readln(Name);
      Result := D_ScanTitle(Name, Volume, Title, Typ, Seg_Flag);
      Writeln('parsed: ');
      case result of
      d_okay:begin
                  Writeln('  Volume name   -- ', Volume);
                  Writeln('  File name     -- ', Title);
                  Write   ('  File type     -- ');
                  Case Typ Of
                     D_Text : Writeln('text file');
                     D_Code : Writeln('code file');
                     D_Data : Writeln('data file');
                     D_SVol : Writeln('svol file');
                  End; { Cases }
                  If Seg_Flag <> 0 Then
                     Writeln('  Segment flag -- ', Seg_Flag);
               end;
      d_name_error:writeln(' Name error');
      end;
      Writeln;
      Write('Continue? ');
      Read(Ch);
      Writeln;
   Until Ch In ['n', 'N'];
End. { Scan_Test }
```

```
Function D_Dir_List (D_NAME:String;
                     D_SELECT    : D_Choice;
                     Var D_PTR   : D_ListP;
                     D_PINFO     : Boolean) : D_Result;
```

D_Dir_List creates a list of records containing directory information on volumes and disk files. This information includes volume names and unit numbers of blocked and unblocked on-line units, number of files on blocked units, lengths and starting blocks of disk files and unused disk spaces, file names and types, and file dates. The function result value indicates invalid file name arguments, off-line volumes, or not-found files.

D_Dir_List optionally provides information describing how the wild card file name argument matched files and/or volumes.

Parameters and Function Results:

D_Dir_List accepts a set specifying the file types on which to return information, and a string containing a file name. D_Dir_List returns a pointer to a linked list of directory information records. Each record contains the name of a file or volume which matches the file name argument and also is one of the types specified in the file type set.

D_NAME--The D_NAME parameter contains a file name which may contain wild cards.

D_SELECT--The D_SELECT parameter is a set specifying the directory objects for which information is to be returned by D_Dir_List. See the file type selection for more information on directory object selection.

D_PTR--The D_PTR parameter is assigned a pointer to a linked list of records containing directory information for all specified file objects. In order to have information returned describing it, a file object must meet the following criteria:

It must reside on a volume which matches the volume ID in D_NAME.

If the object is a disk file, it must match the file ID in D_NAME.

It must belong to one of the types included in D_SELECT.

The linked list contains one record for each file object matched. The records are defined as follows:

```
D_ListP = ^D_List;
D_List = Record
            D_Unit : Integer;
            D_Volume : String[7];
            D_VPat : D_PatRecP;
            D_NextEntry : D_ListP;
            Case D_IsBlkd : Boolean Of
              True : (D_Start,
                        D_Length : Integer;
                        Case D_Kind : D_NameType Of
                          D_Vol,
                          D_Temp,
                          D_Code,
                          D_Text,
                          D_Data,
                          D_SVol:

                            (D_Title : String[15];
                            D_FPat   : D_PatRecP;
                            D_Date   : D_DateRec;
                            Case D_NameType of
                              D_Vol:(D_NumFiles:Integer)));
         End;
```

The fields in the D_List record return the following information for each file object in the D_Ptr list:

D_Unit returns the unit number of the unit containing the object.

D_Volume returns the name of the volume containing the object.

D_VPat is a pointer to pattern matching information collected while comparing volumes to the volume ID in D_NAME (see the wild card UNIT for details on pattern matching info). D_VPat is set to NIL if pattern matching information is not requested.

D_NextEntry is a pointer to the next directory information record in the list. It is set to NIL if the current record is the last record in the list.

D_IsBlkd is set to TRUE if the file object is (or resides on) a block-structured unit. Records describing serial volumes have D_IsBlked set to FALSE; the remaining fields are undefined.

The following fields exist only in records describing file objects stored on disk units (i.e. D_IsBlkd is TRUE):

D_Start contains the starting block number of the file object. If the object is of type D_Vol, this value is interpreted as the block number of the first block on the volume (e.g. 0 for disk volume).

D_Length contains the length (in blocks) of the file object. If the object is of type D_Vol, this value is interpreted as the total number of blocks on the volume (e.g. 494 for single density 8" floppy disk).

D_Kind indicates the type of the file object described by the current record.

The following fields exist only in records describing disk file objects other than unused disk areas (i.e. D_Kind in [D_Vol, D_Temp, D_Code, D_Text, D_Data, D_SVol]):

D_Title contains the file title of the object. For objects of type D_Vol, this field contains the empty string.

D_FPat is a pointer to pattern matching information collected while comparing file names to the file ID in D_NAME (see wild card UNIT for details on pattern matching info). D_FPat is set NIL if pattern matching information is not requested or if the file ID in D_NAME is empty.

D_Date contains the file date for the current object.

D_NumFiles is valid only for objects of type D_Vol; it contains the number of files in the volume's directory.

NOTE: An .SVol file (which contains a subsidiary volume) appears as any other file on the principal volume. This means that D_NumFiles does not correspond to an .SVol file. However, the actual subsidiary volume, when accessed by its volume ID, will return with a valid D_NumFiles entry.

File information is returned (in a linked list accessed by D_Ptr) in the following order:

1. Volume on highest numbered unit which matches D_NAME (if D_Vol is in D_SELECT)

2. Files in directory of this volume which match D_NAME and are of one of the types in D_SELECT (if a file type is in D_SELECT)

   Last file on volume
   .
   .
   First file on volume

3. Unused spaces on this volume (if D_Free is in D_SELECT)

   Last free space on volume
   .
   .
   First free space on volume

4. Volume on lowest numbered unit which matches D_NAME (if D_Vol is in D_SELECT)

5. Files in directory of this volume which match D_NAME and are of one of the types in D_SELECT (if a file type is in D_SELECT)

   Last file on volume
   .
   .
   First file on volume

6. Unused spaces on this volume (if D_Free is in D_SELECT)

   Last free space on volume
   .
   .
   First free space on volume

D_PINFO

When set to TRUE, the D_PINFO parameter indicates that pattern matching information should be returned in a linked list accessed by D_PTR. This information is collected by the D_WILD_MATCH function in the process of comparing volume and fileID's, and is useful for determining how the wild cards in D_NAME were expanded. Information is returned in two pointers; one for volume names matched (named D_VPat) and one for file ID's matched (named D_FPat).

Example of pattern record lists:

D_NAME is set to '=:TEST{1-9}='

Two volumes contain files which match D_NAME:

BOOT contains TEST5.CODE
WORK contains TEST5.TEXT

For BOOT:TEST5.CODE, D_Volume is 'BOOT', D_Title is 'TEST5.CODE', and D_VPat returns a pointer to the following information:

1. WildPos is 1, WildLen is 1
   CompPos is 1, CompLen is 4
   ('=' matches 'BOOT')

D_FPat returns a pointer to the following information:

1. WildPos is 1, WildLen is 4
   CompPos is 1, CompLen is 4
   ('TEST' matches 'TEST')

2. WildPos is 5, WildLen is 5
   CompPos is 5, CompLen is 1
   ('{1-9}' matches '5')

3. WildPos is 10, WildLen is 1
   CompPos is 6, CompPos is 5
   ('=' matches '.CODE')

A similar list is returned for WORK:TEST5.TEXT.

NOTE: If the volume ID in D_NAME consists of a unit number (e.g. "#5"), the volume assigned to the unit is defined to match the volume ID in D_NAME. The Pos and Len pointers are set as in the following example:

D_NAME is set to "#5:"

A disk volume named "FOON" resides in unit 5.

1. WildPos is 1, WildLen is 2
   CompPos is 1, CompPos is 4
   ('#5' matches 'FOON')

NOTE: D_FPat and D_VPat never contain invalid information. If information is unavailable or has not been requested, the pointers are set to NIL.

Function Result

D_Dir_List returns a value of type D_Result. D_Dir_List can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay--No error. All D_Ptr information is valid.

D_Not_Found--No such file/volume found. No match found for D_NAME. D_Dir_List sets D_Ptr to NIL.

D_Name_Error--Illegal syntax in D_NAME. D_Dir_List sets D_Ptr to NIL.

D_Off_Line--Volume/unit off-line. The volume specified by D_NAME was not on-line. This error occurs only when the volume ID in D_NAME does not contain wild cards (i.e. a single volume is specified, and it is off-line). If the volume name in D_NAME contains wild cards but does not match any on-line volumes, D_Dir_List returns D_Not_Found. D_Ptr is set to NIL.

D_Other--Unknown error. D_Dir_encountered an error it could not identify, but which interrupted normal execution of the function. D_Ptr is set to NIL.

Example Program:

The following program is a general purpose directory lister; it accepts a string containing wild cards and creates a list of matching files and (if requested) pattern matching information for the files. Note that the program uses the MARK and RELEASE intrinsics to remove the D_Dir_List information from the heap after the information has been used.

```
Program Listtest;
Uses
    (*$UWILD.CODE*)
    wild,
    (*$UDIR.INFO.CODE*)
    Dirinfo;

Var
    Select : D_Choice;
    Want_Patterns : Boolean;
    Heap_Ptr : ^Integer;
    Segs : Integer;
    Typ : D_NameType;
    Volume, Title, Match : String;
    Result : D_Result;
    Ch : Char;
    Ptr : D_ListP;

    Procedure GiveChoice(Choice : String; Kind : D_Choice);
    Var
        Ch : Char;

    Begin
        Write('       ',Choice,' ? ');
        Read(Ch); Writeln;
        If Ch In ['y', 'Y'] Then Select := Select + Kind;
    End; { GiveChoice }

    Procedure Print_Patterns(PatPtr : D_PatRecP;
                                      Comp, Wild : String);
    Var
        Count : Integer;

    Begin { Print_Patterns }
        Count := 1;
        Writeln('type <cr> for patterns');
        Readln; Writeln;
        Repeat
            Writeln('Pattern ', Count, ' :');
            With PatPtr^ Do
                Begin
                    Writeln('  Comp : ', Comp);
                    If CompLen <> 0 Then
                       Write('^':(CompPos + 9));
                    If CompLen > 1 Then Write('^':(CompLen - 1));
```

```
                Writeln;
                Writeln('  Wild : ', Wild);
                Write('^':(WildPos + 9));
                If WildLen > 1 Then Write('^':(WildLen - 1));
                Writeln; Writeln;
            End;
        PatPtr := PatPtr^.Next;
        Count := Count + 1;
    Until PatPtr = Nil
End; { Print_Patterns }

Procedure Print_Info(Ptr : D_ListP);

Begin { Print_Info }
    Repeat
        With Ptr^ Do
            Begin
                If D_IsBlkd Then
                  Case D_Kind Of
                      D_Free : Write('Free space on ');
                      D_Vol  : Write('Volume ');
                      D_Temp : Write('Temporary file on ');
                      D_Text : Write('Text file on ');
                      D_Code : Write('Code file on ');
                      D_Data : Write('Data file on ');
                      D_SVol : Write('SVol file on ');
                  End { Cases }
                Else
                  Write('Unblocked volume ');
                Writeln(D_Volume);
                If Want_Patterns And (D_VPat <> Nil) Then
                  Begin
                      Writeln;
                      Writeln('      Volume patterns:');
                      Print_Patterns(D_VPat, D_Volume, Volume);
                  End;
                Writeln('      Unit number ..... ', D_Unit);
                If D_IsBlkd Then
                  Begin
                      If Not (D_Kind In [D_Vol, D_Free]) Then
                        Writeln('      File name ..... ', D_Title);
                      If D_Kind <> D_Free Then
                        Begin
                          If Want_Patterns And (D_FPat <> Nil) Then
                            Begin
                              Writeln('      File name patterns:');
```

```
                                Print_Patterns(D_FPat, D_Title, Title);
                              End;
                            With D_Date Do
                              Writeln('      File date ..... ',
                                      Month, '/', Day, '/', Year);
                          End; { If D_Kind }
                        If D_Kind = D_Vol Then
                          Writeln('  Files on volume ... ', D_NumFiles);
                        Writeln('      Starting block .... ', D_Start);
                        Writeln('      File length ..... ', D_Length);
                    End; { If D_IsBlkd }
                End; { With Ptr^ }
            Writeln;
            Write('Type <cr> for rest of list');
            Readln; Writeln;
            Ptr := Ptr^.D_NextEntry;
        Until Ptr = Nil
    End; { Print_Info }

Begin { D_Test }
    Repeat
        Mark(Heap_Ptr);
        Select := [];
        Writeln('Directory Lister --');
        Write('Volume and/or file name to match: ');
        Readln(Match);
        Write('Return pattern matching information? [y/n] ');
        Read(Ch); Writeln;
        Want_Patterns := Ch In ['y', 'Y'];
        If Want_Patterns Then
            Result := D_ScanTitle(Match, Volume, Title, Typ, Segs);
        Writeln('Types [ y/n ] : ');
        GiveChoice('Directories', [D_Vol]);
        GiveChoice('Text Files ', [D_Text]);
        GiveChoice('Code Files ', [D_Code]);
        GiveChoice('Data Files ', [D_Data]);
        GiveChoice('Temp Files ', [D_Temp]);
        GiveChoice('Free Space ', [D_Free]);
        GiveChoice('SVol Files ', [D_SVol]);
        Result := D_DirList(Match, Select, Ptr, Want_Patterns);
        Writeln;
        If Ptr <> Nil Then
            Print_Info(Ptr)
        Else
            Case Result Of
                D_Name_Error : Writeln('      Error in file name');
```

```
                D_Off_Line : Writeln('      Volume off line');
                D_Not_Found : Writeln('     File not found');
                D_Other : Writeln('     Miscellaneous error');
            End; {cases}
        Writeln;
        Repeat
            Write('Continue ? ');
            Read(Ch); Writeln;
        Until Ch In ['n','N','y','Y'];
        Writeln;
        Release(Heap_Ptr);
    Until Ch In ['n', 'N'];
End. { listtest }
```

Function D_Change_Name (D_OLD_NAME,
D_NEW_NAME : String;
D_REMOLD : Boolean) : D_Result;

D_Change_Name searches for the volume or file designated by the file name contained in D_OLD_NAME and changes its name to the file name contained in D_NEW_NAME.

D_Change_Name only changes one file name at a time, and thus does not accept file names containing wild cards; however, it can be combined with other Dir_Info and wild card routines to create user-defined file name changing routines which accept wild cards.

Parameters and Function Results:

D_Change_Name accepts the following parameters:

D_OLD_NAME--A string containing the name of the file to be changed. If the file name is invalid, D_Change_Name returns D_Name_Error. Note that wild card characters are treated literally.

D_NEW_NAME--A string containing the replacement file name. If the file name is invalid, D_Change_Name returns D_Name_Error. Note that wild card characters are treated literally.

If D_OLD_NAME contains an empty file title, D_Change_Name changes the name of the volume specified by D_OLD_NAME to the volume name in D_NEW_NAME; any file title in D_NEW_NAME is ignored. If D_OLD_NAME contains an nonempty file title, D_Change_Name changes the name of the disk file specified by D_OLD_NAME to the file title in D_NEW_NAME; any volume name in D_NEW_NAME is ignored. If the file ID in D_NEW_NAME is empty, D_Change_Name returns D_Name_Error.

D_REMOLD--If set to TRUE, D_REMOLD indicates that an existing file or volume designated by the file name in D_NEW_NAME may be removed in order to change the file name. If set to FALSE, the presence of an existing file or volume with the same name as D_NEW_NAME aborts the name change, and D_Change_Name returns D_Exists as a function result.

D_Change_Name returns a value of type D_Result. D_Change_Name can return all scalar values defined in D_Result; the values have the following meanings:

D_Okay--No error. D_OLD_NAME was found and its name changed.

D_Not_Found--No such file/volume found. No match found for D_OLD_NAME. No change made.

D_Exists--The name change was blocked by the presence of an existing file with the same name as D_NEW_NAME. No change made.

D_Name_Error--Illegal file name syntax in D_OLD_NAME or D_NEW_NAME. No change made.

D_Off_Line--Volume/unit off-line. Volume/unit specified by D_OLD_NAME was not on-line. No change made.

D_Other--Unknown. D_Change_Name encountered an error it could not identify. No change made.

Programming Example:

The following program demonstrates the use of D_Change_Name.

```
Program chngtest;
Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  DirInfo;

Var
  RemOld : Boolean;
  Old, New : String;
  Ch : Char;
  Rslt : D_Result;

Begin { chngtest}
  Writeln('D_ChangeName Test --');
```

```
  Repeat
    Writeln;
    Write('Name to change : ');
    Readln(Old);
    Write('New name : ');
    Readln(New);
    Write('Remove existing files (if any) of that name ? [y/n] ');
    Read(Ch); Writeln;
    RemOld := Ch In ['y','Y'];
    Case D_ChangeName(Old,New,RemOld) Of
      D_Okay : Writeln('       No error');
      D_Off_Line : Writeln('       Volume off line');
      D_Name_Error : Writeln('       Error in file name');
      D_Not_Found : Writeln('       File not found');
      D_Other : Writeln('       Miscellaneous error');
    End; { cases }
    Writeln;
    Write('Continue ? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End. { chngtest }
```

Wild Card File Name Replacement

D_Change_Name does not accept wild card file name arguments; however, it can be combined with the pattern matching information returned by D_Dir_List to implement a wild card, file name changing routine. (Note that this routine must use directory locks in multi-tasking environments.)

For example, assume that the user has the following files:

```
TEST1.TEXT
TEST12.CODE
TEST.DATA
```

The user would like to change them to:

```
OLD1A.TEXT
OLD12A.CODE
OLDA.DATA
```

This can be performed by using D_Dir_List to search for the file name 'TEST=.='. The pattern matching information returned by D_Dir_List can be used to create new file titles; in this case, 'TEST' is replaced with 'OLD', and the first '=' is replaced with the catenation of the pattern matched by the '=' and the literal string 'A'. The part of each file title matched by the period and the second "=" wild card is unchanged. D_Change_Name is called with the modified file title for each file matched by D_Dir_List.

Programming Example:

The following program demonstrates the use of D_Change_Name and D_Dir_List in the construction of a specialized file name changing utility. The program accepts a file name argument containing two '=' wildcards; for each file which matches the argument, the file title is changed by swapping the string patterns matched by the two "=" wildcards.

```
Program WildChng;

Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  DirInfo;

Var
  Heap_Ptr : ^Integer;
  Typ : D_NameType;
  Segs : Integer;
  Select : D_Choice;
  Volume, Name, Match : String;
  Result : D_Result;
  Ch : Char;
  Ptr : D_ListP;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var
    Ch : Char;

  Begin
    Write('       ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }

  Procedure Print_Patterns(PatPtr : D_PatRecP;
```

```
                                        Comp, Wild : String);
  Var
    Count : Integer;

  Begin { Print_Patterns }
    Count := 1;
    Writeln('type <cr> for patterns');
    Readln; Writeln;
    Repeat
      Writeln('Pattern ', Count, ' :');
      With PatPtr^ Do
        Begin
          Writeln('  Comp : ', Comp);
          If CompLen <> 0 Then
            Write('^':(CompPos + 9));
          If CompLen > 1 Then Write('^':(CompLen - 1));
          Writeln;
          Writeln('  Wild : ', Wild);
          Write('^':(WildPos + 9));
          If WildLen > 1 Then Write('^':(WildLen - 1));
          Writeln; Writeln;
        End;
      PatPtr := PatPtr^.Next;
      Count := Count + 1;
    Until PatPtr = Nil
  End; { Print_Patterns }

Procedure Print_Info(Ptr : D_ListP; Want_Patterns : Boolean;
                                          Volume, Name : String);
  Begin { Print_Info }
    Repeat
      Writeln('MATCHED FILE --');
      With Ptr^ Do
        Begin
          Write(D_Volume, ':');
          If D_IsBlkd Then
            If Length(D_Title) > 0 Then
              Write(D_Title);
          Writeln;
          If Want_Patterns And (D_VPat <> Nil) Then
            Begin
              Writeln;
              Writeln('       Volume patterns:');
              Print_Patterns(D_VPat, D_Volume, Volume);
            End;
          If D_IsBlkd Then
```

```
            If Want_Patterns And (D_FPat <> Nil) Then
              Begin
                Writeln('       File name patterns:');
                Print_Patterns(D_FPat, D_Title, Name);
              End;
        End; { With Ptr^ }
      Writeln;
      Write('Type <cr> for rest of list');
      Readln; Writeln;
      Ptr := Ptr^.D_NextEntry;
    Until Ptr = Nil
End; { Print_Info }

Procedure Change(Ptr : D_ListP; Name : String);
Var
  I, Pos1, Len1, Pos2, Len2, Last_Pos,
  Mid_Pos, Last_Equal : Integer;
  Pat1, Pat2, Title, New : String;

  Procedure Find_Equal(D_Title, Name : String;
                       Var PatPtr : D_PatRecP;
                       Var Pat : String;
                       Var Pos, Len : Integer);

  Begin { Find_Equal }
    While (Name[PatPtr^.WildPos] <> '=') And
      (PatPtr^.Next <> Nil) Do
        PatPtr := PatPtr^.Next;
    With PatPtr^ Do
      Begin
        If CompLen = 0 Then Pat := ''
        Else Pat := Copy(D_Title, CompPos, CompLen);
        Pos := CompPos;
        Len := CompLen;
      End;
  End; { Find_Equal }

Begin { Change }
  With Ptr^ Do
    Begin
      Find_Equal(D_Title, Name, D_FPat, Pat1, Pos1, Len1);
      If D_FPat <> Nil Then
        Begin
          D_FPat := D_FPat^.Next;
          Find_Equal(D_Title, Name, D_FPat, Pat2, Pos2, Len2);
          New := D_Title;
```

```
            Last_Pos := Pos2 + Len2;
            Mid_Pos := Pos1 + Len2;
            Last_Equal := Last_Pos - Len1;
            For I := Pos1 To Mid_Pos - 1 Do { 1st '=' }
              New[I] := Pat2[I - Pos1 + 1];
            For I := Mid_Pos To Last_Equal - 1 Do
              New[I] := D_Title[I - Len2 + Len1];
            For I := Last_Equal To Last_Pos - 1 Do { 2nd '=' }
              New[I] := Pat1[I - Last_Equal + 1];
            New := Concat(D_Volume, ':', New);
            Title := Concat(D_Volume, ':', D_Title);
            Result := D_ChangeName(Title, New, True);
            Write(Title, '-->', New);
            Case Result Of
              D_Name_Error : Write('  Error in file name');
              D_Off_Line : Write('  Volume off line');
              D_Not_Found : Write('  File not found');
              D_Other : Write('  Miscellaneous error');
            End; {cases}
            Writeln;
          End; { if D_FPat }
      End; { with }
  End; { Change }

  Function Display(S, Match, Volume, Name : String;
                   Select : D_Choice) : D_ListP;
  Var
    Ch : Char;
    Ptr : D_ListP;
    Want_Patterns : Boolean;
    Result : D_Result;

  Begin { Display }
    Writeln; Writeln(S);
    Write('      Display pattern matching information ? ');
    Read(Ch); Writeln;
    Want_Patterns := Ch In ['y', 'Y'];
    Result := D_DirList(Match, Select, Ptr, True);
    If Ptr <> Nil Then
      Print_Info(Ptr, Want_Patterns, Volume, Name)
    Else
      Case Result Of
        D_Name_Error : Writeln('      Error in file name');
        D_Off_Line : Writeln('      Volume off line');
        D_Not_Found : Writeln('      File not found');
        D_Other : Writeln('      Miscellaneous error');
```

```
      End; {cases}
    Display := Ptr;
  End; { Display }


Begin { WildChange }
  Writeln;
  Repeat
    Mark(Heap_Ptr);
    Select := [];
    Write('File title to match (must contain two ''='' ): ');
    Readln(Match);
    Result := D_ScanTitle(Match, Volume, Name, Typ, Segs);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    GiveChoice('SVol Files ', [D_SVol]);
    Ptr := Display('Old Files :', Match, Volume, Name, Select);
    If Ptr <> Nil Then
      Begin
        Repeat
          Change(Ptr, Name);
          Ptr := Ptr^.D_NextEntry;
        Until Ptr = Nil;
        Write('Redisplay files? ');
        Read(Ch); Writeln;
        If Ch In ['y', 'Y'] Then
          Ptr := Display('New Files :', Match,
                         Volume, Name, Select);
      End;
    Writeln;
    Repeat
      Write('Continue ? ');
      Read(Ch); Writeln;
    Until Ch In ['n','N','y','Y'];
    Writeln;
    Release(Heap_Ptr);
  Until Ch In ['n', 'N'];
End. { WildChng }
```

Function D_Change_Date (D_NAME : String;
D_NEWDATE : D_DateRec;
D_SELECT : D_Choice) : D_Result;

D_Change_Date changes the file date of volumes and files whose names match the file name argument contained in D_NAME. D_Change_Date accepts wild cards in its file name argument. If a volume date is changed, only the disk is updated. The disk must be re-booted if the new date is to be used. In order to change the internal date (which will appear when D(ate is used in the filer) use the date access procedures within the SYS.INFO unit.

Parameters and Function Results:

D_Change_Date accepts the following parameters:

D_NAME--A string which contains a valid file name. The file name may contain wild cards.

D_NEWDATE--A record of type D_DateRec which contains the new date. A year value of 100 is not accepted by D_Change_Date in a new date.

D_SELECT--A set of file and/or volume types as described in section 2.2. All scalar types except D_Free and D_Temp apply to D_Change_Date. Disk free spaces identified by the D_Free scalar do not contain file dates. Temporary status for files is specified by a special value in the file date field. Thus, D_Free and D_Temp are ignored if they are included in D_SELECT.

D_Change_Date returns a value of type D_Result. D_Change_Date can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay--No error. D_NAME was found, and D_NEWDATE was written to the directory for the specified file or disk volume.

D_Not_Found--No such file/volume found. No match found for D_NAME. No change made.

D_Name_Error--Illegal syntax in D_NAME. No change made.

D_Off_Line--Volume/unit off-line. Volume/unit specified by D_NAME was not on-line. No change made. This error occurs only if the volume ID in D_NAME specifies a single volume which is off-line. If the volume name in D_NAME contains wild cards and does not match any on-line volumes, D_Change_Date returns D_Not_Found.

D_Other--Unknown error. No change made. D_Change_Date encountered an unidentified error which prevented successful completion of the operation.

Programming Example:

The following program demonstrates the use of D_Change_Date.

```
Program Date_Test;
Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  DirInfo;

Var
  Result    : D_Result;
  Ch        : Char;
  M, D, Y   : Integer;
  NewDate   : D_DateRec;
  Select    : D_Choice;
  FileName  : String;

  Procedure GiveChoice(Choice : String; Kind : D_Choice);
  Var
    Ch : Char;
  Begin
    Write('       ',Choice,' ? ');
    Read(Ch); Writeln;
    If Ch In ['y', 'Y'] Then Select := Select + Kind;
  End; { GiveChoice }

Begin { Date_Test }
  Select := [];
  Writeln('D_ChangeDate Test --');
  Repeat
    Writeln;
    Write('File to change : '); Readln(FileName);
    Writeln('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
```

```
      GiveChoice('Code Files ', [D_Code]);
      GiveChoice('Data Files ', [D_Data]);
      GiveChoice('SVol Files ', [D_SVol]);
      Writeln('New date : ');
      Write('Month [1 - 12] : '); Readln(M);
      Write('Day   [1 - 31] : '); Readln(D);
      Write('Year  [0 - 99] : '); Readln(Y);
      With NewDate Do
        Begin
          Month := M;
          Day := D;
          Year := Y;
        End; { With NewDate }
      Writeln;
      Result := D_ChangeDate(FileName, NewDate, Select);
      Case Result Of
        D_Okay : Writeln('date changed');
        D_Name_Error : Writeln('error in file name');
        D_Off_Line : Writeln('volume off line');
        D_Not_Found : Writeln('file not found');
        D_Other : Writeln('miscellaneous error');
      End; { cases }
      Writeln;
      Write('Continue ? ');
      Read(Ch); Writeln;
    Until Ch In ['n','N'];
End. { Date_Test }
```

Function D_Rem_Files (D_NAME : String;
D_SELECT : D_Choice) : D_Result;

The D_Rem_Files function removes file objects whose names match the file name argument contained in D_NAME and types match the elements included in D_SELECT. The file name argument may contain wildcards. Disk files are permanently deleted from their directories. Volumes are taken off-line, but not altered in any way; off-line disk volumes may be brought back on-line merely by referencing them, while off-line serial volumes remain inaccessible until the system is reinitialized.

Parameters and Function Result:

D_Rem_Files accepts the following parameters.

D_NAME--A string containing the name of the file(s) or volume(s) to be removed.

D_SELECT--A set of file objects to be removed. Removal of file objects is subject to the criteria described in section 4.2.0.2. The definition of the set is as follows:

```
D_NameType = (D_Vol, D_Code, D_Text, D_Data, D_SVol, D_Temp, D_Free);

D_Choice   = Set Of D_NameType;
```

All scalar types except D_Free apply to D_Rem_Files. Disk free space cannot be removed from the directory; thus, D_Free is ignored if it is included in D_SELECT.

D_Rem_Files returns a value of type D_Result. D_Rem_Files can return all scalar values defined in D_Result except D_Exists; the values have the following meanings:

D_Okay--No error. D_NAME was found. If D_Vol is included in D_SELECT, and a volume matches the file name argument in D_NAME, the volume is taken off-line. If D_Text, D_Code, D_Data, D_SVol, or D_Temp are included in D_SELECT, disk files of those types which match D_NAME are deleted from their directories.

D_Not_Found--No such file/volume found. No match found for D_NAME. No change made.

D_Name_Error--Illegal file name syntax in D_NAME. No change made.

D_Off_Line--Volume/unit off-line. The volume/unit specified by D_NAME was not on-line. No change made. This error occurs only if the volume id in D_NAME specifies a single volume which is off-line. If the volume id in D_NAME contains wildcards, but does not match any on-line volume, D_Rem_Files returns D_Not_Found.

D_Other--Unknown error. No change made. D_rem_Files encountered an unidentified error which prevented successful completion of the operation.

Programming Example:

```
Program Rem_Test;
Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  Dirinfo;
```

```
Var
   Result : D_Result;
   Select : D_Choice;
   Ch : Char;
   Remfile : String;

   Procedure GiveChoice(Choice : String; Kind : D_Choice);
   Var
      Ch : Char;

   Begin
      Write('       ',Choice,' ? ');
      Read(Ch); Writeln;
      If Ch In ['y', 'Y'] Then Select := Select + Kind;
   End; { GiveChoice }

Begin { Rem_Test }
   Select := [];
   Writeln('D_RemFiles Test --');
   Repeat
      Write('File(s) to remove : ');
      Readln(Remfile);
      Writeln('Types [ y/n ] : ');
      GiveChoice('Directories', [D_Vol]);
      GiveChoice('Temp Files ', [D_Temp]);
      GiveChoice('Text Files ', [D_Text]);
      GiveChoice('Code Files ', [D_Code]);
      GiveChoice('Data Files ', [D_Data]);
      GiveChoice('SVol Files ', [D_SVol]);
      Result := D_RemFiles(Remfile, Select);
      Case Result Of
         D_Okay : Writeln('files removed');
         D_Name_Error : Writeln('error in file name');
         D_Off_Line : Writeln('volume off line');
         D_Not_Found : Writeln('file not found');
         D_Other : Writeln('miscellaneous error');
      End; { cases }
      Writeln;
      Write('Continue ? ');
      Read(Ch); Writeln;
   Until Ch In ['n','N'];
End. { Rem_Test }
```

Procedure D_Lock;

D_Lock grants exclusive directory access rights to the task that executes it; however, a task may have to wait until another task releases the directory lock before it can continue execution past its call to D_Lock.

NOTE: D_Lock calls should always be matched with D_Release calls to prevent system deadlocks.

The Dir_Info routines D_Lock and D_Release are provided for use in multi-tasking environments. When used properly, they ensure mutually exclusive access to directory information.

Procedure D_Release;

D_Release releases exclusive access rights to the directory. Tasks already waiting for directory access are automatically awakened when the directory becomes available by a call to D_Release.

Programming Example:

The following program demonstrates the use of D_Lock and D_Release.

```
Program Locktest;
Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  DirInfo;

Const
  Stack_Size = 2000;

Var
  Pid : Processid;
  Old,
  New : String;
  Date : D_DateRec;
  M, D, Y : Integer;
  Ch : Char;

  Process Change_And_Check(Old, New : String; Date : D_DateRec);
  Var
    Result : D_Result;
```

```
  Begin { Change_And_Check }
    D_Lock;              { beginning of critical section }
    Result := D_ChangeDate(Old, Date, [D_Vol..D_SVol]);
    If Result = D_Okay Then
      Result := D_ChangeName(Old, New, True);
    D_Release;           { end of critical section }
  End; { Change_And_Check }

Begin { LockTest }
  Repeat
    Write('Old file name: ');
    Readln(Old);
    Write('New file name: ');
    Readln(New);
    Writeln('New date:');
    Write('    Month: ');
    Readln(M);
    Write('      Day: ');
    Readln(D);
    Write('     Year: ');
    Readln(Y);
    With Date Do
      Begin
        Month := M;
        Day := D;
        Year := Y;
      End;
    Start(Change_And_Check(Old, New, Date), Pid, Stack_Size);
    Write('Start another? ');
    Read(Ch); Writeln;
  Until Ch In ['n', 'N'];
End. {Locktest }
```

II.3.6.2 Wild Cards (WILD)

The unit WILD provides a wild card convention for pattern matching of string variables. Wild cards are special character sequences in a character string; they are named wild cards because of their ability to match whole classes of character sequences rather than a single character sequence. For instance, the string "a=" matches all character strings starting with the letter "a" because "=" is defined as a wild card which matches any character sequence.

Wild cards are useful in pattern matching situations where many character strings are to be matched with a single request. The p-System Filer uses a set of wild card facilities in its directory operations. Examples are given in the p-System manual that describes the filer operation. Because of the extra functions provided by this UNIT there is not a direct correspondence between the filer and this UNIT. Where there are differences in the use of characters these are described.

II.3.6.2.1 Special Wild Card Characters

The following characters are defined as special characters:

question mark	?
equals sign	=
braces	{ and }
comma	,
hyphen	-
tilde	~
percent sign	%

Special characters may only be used as parts of wild cards; however, a literal occurrence of a special character can be represented by a two character sequence consisting of a percent sign followed by the special character. A percent sign indicates that the following character is to appear literally in the character string; for instance, "xx%=yy" is treated as the literal character string "xx=yy" rather than a wild card string.

Examples of percent sign in wild cards:

"a b%?def"	matches	"ab?def"
"ab{a-z, %=}de%%f"	matches	"ab=de%f"
"ab%-def"	matches	"ab-def"

II.3.6.2.1.1 Question Mark Wild Card

A question mark matches any single character. In the filer the "?" is treated as an interactive query of an "=" wild card. This is one of the major differences in use of characters between this UNIT and the filer.

Examples of "?" wild card:

Pattern:	"ab?def"
Matches:	"abbdef" "abrdef"
Nonmatch:	"abdef" "abjkdef" "abef"

II.3.6.2.1.2 Equals Sign Wild Card

An equals sign matches any sequence of characters, including the empty sequence. This is the same as the filer except that more than one "=" can appear in a wild card string.

Examples of "=" wild card:

Pattern:	"ab=def="
Matches:	"abcdefg" "abdef" "abcccdef"
Nonmatches:	"abcef"

II.3.6.2.1.3 Subrange Wild Card

The subrange wild card matches a single character from the character set specified in the subrange. The following special characters are used to construct subrange wild cards: comma, hyphen, tilde, and braces.

A subrange wild card consists of a character set delimited by braces. A character set consists of a list of character-items separated by commas.

A character-item is either a character or a character range (two characters separated by a hyphen). A character range implicitly specifies all characters lying between the two characters. (Consult an ASCII table to determine the ordering of characters.)

Character-items preceded by tildes are called negated-items and are specifically excluded from the character set. A character range proceeded by a tilde is entirely excluded from the character set. The list of character items is evaluated left-to-right. Characters specified by non-negated items are included into the set; characters specified by negated items are excluded from the set. Thus, a character matches the subrange wild card if it matches one of the non-negated items, but does not match any of the negated choices. For example, the subrange "{a-z,~r}" represents the set of characters from "a" to "z", excluding "r".

NOTE: Blank characters within subrange wild cards are ignored. Wild Card characters can be specified in character sets with the percent sign notation described above.

Examples of subrange wild cards:

```
{a,b,c}
{a-d,j,w-z}
{a-z,~j,~x-y}
```

Syntax for subrange wild card:

```
wild-card       =    "{" item-list "}"
item-list       =    item < "," item >
item            =    [~] char-item
char-item       =    char / range
range           =    char "-" char
char            =    a  printable ASCII character
```

Examples of subrange wild card:

Pattern:	"ab{a-r, ~j, ~k}def"
Matches:	"abbdef" "abrdef"
Nonmatches:	"abjdef" "abkdef" "abzdef"

II.3.6.2.2 Function D_Wild_Match (WILD, COMP: String; Var PPTR : D_PatRecP; PINFO : Boolean) : Boolean;

D_Wild_Match serves as a general purpose pattern matcher for string variables using the wild card conventions described above. The two main parameters are a wild card string WILD and a literal string COMP. D_Wild_Match determines whether the literal string matches the wild card string. If the strings match, D_Wild_Match returns TRUE; otherwise, it returns FALSE. If PINFO is set to TRUE, D_Wild_Match returns information (accessed through PPTR) which describes how the strings were matched.

D_Wild_Match Parameters and Function Result:

D_Wild_Match accepts the following parameters.

WILD--A string which may contain wild cards.

COMP--A literal text string.

PINFO--A Boolean. If set to TRUE, PINFO requests that pattern matching information be returned.

PPTR--Pointer of type D_PatRecP. Depending on the value passed in PINFO, D_Wild_Match either sets PPTR to NIL or points it at a linked list of records containing pattern matching information.

II.3.6.2.2.1 D_Wild_Match Pattern Matching Information

If PINFO is set to TRUE, D_Wild_Match returns pattern matching information in PPTR. PPTR is a pointer (of type D_PatRecP) to a linked list of records which contain the starting positions and lengths of corresponding character patterns in WILD and COMP.

D_Pat_RecP is defined as follows:

```
D_PatRecP  = ^D_PatRec;

D_PatRec   = Record
                CompPos,
                CompLen,
                WildPos,
                WildLen:Integer;
                Next:D_PatRecP;
                End; { D_PatRec }
```

CompPos and WildPos are the starting positions of corresponding character patterns in COMP and WILD, respectively. CompLen and WildLen are the pattern lengths. Next points to the next pattern record in the list; it is set to NIL in the last pattern record. The patterns occur in the list in the order in which they were matched in the strings.

If the strings do not match, or the list was not requested (i.e. PINFO is set to FALSE), PPTR is set to NIL.

Example of pattern record list:

```
WILD contains:'=ab{a-m}=f?'
COMP contains:'abcdefg'
```

If PINFO is set to TRUE, pattern record list returned is--

1. WildPos = 1, WildLen = 1
   CompPos = 1, CompLen = 0
   ('=' matches the empty string)

2. WildPos = 2, WildLen = 2
   CompPos = 1, CompLen = 2
   ('ab' matches 'ab')

3. WildPos = 4, WildLen = 5
   CompPos = 3, CompLen = 1
   ('{a-m}' matches 'c')

4. WildPos = 9, WildLen = 1
   CompPos = 4, CompLen = 2
   ('=' matches 'de')

5. WildPos = 10, WildLen = 1
   CompPos = 6, CompLen = 1
   ('f' matches 'f')

6. WildPos = 11, WildLen = 1
   CompPos = 7, CompLen = 1
   ('?' matches 'g')

NOTE: When the "=" wild card in WILD matches an empty string in COMP, CompLen is set to 0 and CompPos is set to the position of the next pattern in COMP (i.e. the position where a nonempty pattern would have occurred). Be sure to check the validity of CompPos indices before using them to reference characters in COMP; otherwise, range errors may occur.

Unit interface

```
unit wild;
interface
type
   D_PatRecP      = ^D_PatRec;
   D_PatRec       = Record
                       CompPos,
                       CompLen,
                       WildPos,
                       WildLen:Integer;
                       Next:D_PatRecP;
                    End; { D_PatRec }

function d_wild_match(wild,comp:string;
                              var pptr:d_patrecp;
                              pinfo:boolean):boolean;
```

Sample Program:

The following program is an example of a string comparison routine which uses D_Wild_Match. The program reads two strings and prints the result of the comparison; if requested, it also prints information describing how the patterns matched.

```
Program Wild_Test;

Uses (*$UWILD.CODE*)
   wild;

Var
   W, C : String;
   Ch : Char;
   PatPtr : D_PatRecP;
   Want_Patterns : Boolean;

   Procedure Print_Patterns(PatPtr : D_PatRecP;
                                    C, W : String);
   Var
      Count : Integer;

   Begin { Print_Patterns }
      Writeln('type <cr> for patterns');
      Readln; Writeln;
      Count := 1;
      Repeat
```

```
        Writeln('Pattern ', Count, ' :');
        With PatPtr^ Do
          Begin
            Writeln('  Comp : ', C);
            If CompLen <> 0 Then Write('^':(CompPos + 9));
            If CompLen > 1 Then Write('^':(CompLen - 1));
            Writeln;
            Writeln('  Wild : ', W);
            Write('^':(WildPos + 9));
            If WildLen > 1 Then Write('^':(WildLen - 1));
            Writeln; Writeln;
          End;
        PatPtr := PatPtr^.Next;
        Count := Count + 1;
      Until PatPtr = Nil;
    End; { Print_Patterns }

Begin { Wild_Test }
  Repeat
    Writeln('--WildCard Check--');
    Write('Wild Card String    : ');
    Readln(W);
    Write('Comparison String : ');
    Readln(C);
    Write('Do you want pattern matching information ? [y/n] ');
    Read(Ch);
    Want_Patterns := Ch In ['y','Y'];
    Writeln; Writeln;
    If D_Wild_Match(W, C, PatPtr, Want_Patterns) Then
      Writeln('A Match')
    Else Writeln('No Match');
    If Want_Patterns And (PatPtr <> Nil) Then
      Print_Patterns(PatPtr, C, W);
    Write('Continue ? [y/n] ');
    Read(Ch);
    Writeln; Writeln;
  Until Ch In ['n', 'N'];
End. { Wild_Test }
```

II.3.6.3 System Information (SYS.INFO)

Unit SYS.INFO provides an easy way to access some of the system global information. SYS.INFO uses KERNEL.CODE in its implementation section. Although it is possible to access KERNEL.CODE directly, there are many variables that are normally not needed. If a user requires a different set, then another unit similar to this one can be easily constructed for the particular situation.

In order to distinguish the variables defined by this unit, they have been prefixed with SI.

This supplement has a description of each of the functions or procedures. The unit interface is then given, and finally an there is an example program showing the various components of the unit in use.

Work Code File Name:

```
Procedure SI_Code_Vid (Var SI_Vol : String);

Procedure SI_Code_Tid (Var SI_Title : String);
```

The preceding procedures return the volume name (SI_Vol) and the file name (SI_Title) of the system work code file.

Work Text File Name:

```
Procedure SI_Text_Vid (Var SI_Vol : String);

Procedure SI_Text_Tid (Var SI_Title : String);
```

The preceding procedures return the volume name (SI_Vol) and the file name (SI_Title) of the system work text file.

System Unit:

```
Function SI_Sys_Unit : Integer;
```

The SI_Sys_Unit function returns an integer function result. The unit number of the unit containing the system volume is returned.

```
Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
```

The preceding procedure returns the volume name (SI_Vol) of the current system volume.

Prefixed Volume Name:

```
Procedure SI_Get_Pref_Vol (Var SI_Vol : String);

Procedure SI_Set_Pref_Vol (SI_Vol : String);
```

The preceding procedures allow the current prefix volume to be read and set.

System Date:

```
Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);

Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
```

The SI_Get_Date and SI_Set_Date procedures access and modify the system date. The date is passed as a record of type SI_Date_Rec. Changing the date will not change the date on the system disk. It will only change the date internally in the operating system. To change the date on the disk, use function D_Change_Date within the DIR.INFO unit.

```
SI_Date_Rec = Packed Record
                    Month : 0..12;
                    Day : 0..31;
                    Year : 0..99;
              End;
```

This record is used in the operating system to store dates. It is a packed record and only requires 16 bits. All date variables use this format.

Unit interface:

```
 Unit Sys_Info;
Interface
Type
SI_Date_Rec = Packed Record
                        Month : 0..12;
                        Day : 0..31;
                        Year : 0..99;
                        End;

Procedure SI_Code_Vid (Var SI_Vol : String);
Procedure SI_Code_Tid (Var SI_Title : String);
Procedure SI_Text_Vid (Var SI_Vol : String);
Procedure SI_Text_Tid (Var SI_Title : String);
Function  SI_Sys_Unit : Integer;
Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
Procedure SI_Get_Pref_Vol (Var SI_Vol : String);
Procedure SI_Set_Pref_Vol (SI_Vol : String);
Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);
Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
```

Example Program:

```
Program Sys_Test;

Uses {$USys.Info.Code} Sys_Info;

Var
   Ch : Char;
   Date : SI_Date_Rec;
   Vol,
   Title : String;

Begin
   SI_Code_Vid (Vol);
   SI_Code_Tid (Title);
   If Length (Title) <> 0 Then
      Writeln ('The Work Codefile is ', Vol, ':', Title)
   Else
      Writeln ('There is no Work Codefile.');
   SI_Text_Vid (Vol);
   SI_Text_Tid (Title);
   If Length (Title) <> 0 Then
      Writeln ('The Work Textfile is ', Vol, ':', Title)
   Else
```

```
    Writeln ('There is no Work Textfile.');

  Writeln;
  SI_Get_Sys_Vol (Vol);
  Writeln ('The System was booted on volume ', Vol,
           ': on unit ', SI_Sys_Unit);
  SI_Get_Pref_Vol (Vol);

  Writeln;
  Writeln ('The Prefix volume is ', Vol, ':');
  Write ('New Prefix: ');
  Readln (Vol);
  Delete (Vol, Pos (':', Vol), 1);
  If Length (Vol) In [1..7] Then
    Begin
      SI_Set_Pref_Vol (Vol);
      SI_Get_Pref_Vol (Vol);
      Writeln ('The Prefix volume is ', Vol, ':');
    End {of If}
  Else
    Writeln ('No change made');

  Writeln;
  SI_Get_Date (Date);
  Writeln ('The current date is ',
           Date.Month, -Date.Day, -Date.Year);
  Repeat
    Write ('Set for tomorrow''s date ? ');
    Read (Ch);
  Until Ch In ['y', 'Y', 'n', 'N'];
  Writeln;
  If Ch In ['y', 'Y'] Then
    Begin
      Date.Day := Date.Day + 1;
      If (Date.Month In [1, 3, 5, 7, 8, 10, 12]) And (Date.Day = 32) Or
         (Date.Month In [4, 6, 9, 11]) And (Date.Day = 31) Or
         (Date.Month = 2) And (Date.Day = 29) Then
        Begin
          Date.Day := 1;
          If Date.Month = 12 Then
            Begin
              Date.Year := Date.Year + 1;
              Date.Month := 1;
            End {of If =12}
          Else
            Date.Month := 1;
```

```
        End {of If Date.Month};
      SI_Set_Date (Date);
      SI_Get_Date (Date);
      Writeln ('The new date is ',
                Date.Month, -Date.Day, -Date.Year);
    End {of If Ch}
  Else
    Writeln ('No change made');
End {of Sys_Test}.
```

II.3.6.4 File Information (FILE.INFO)

This unit provides an easy way to access information in the file information block (fib). It uses the system globals from KERNEL.CODE. Although it is possible for a user to access this global data, it is easier to use this unit. In order to distinguish the variable names in this unit they have all been prefixed with an F.

The types, functions and procedures defined by the unit are first described. The interface section is then given. Finally, there is an example program showing the various procedures and functions.

Type F_File_Type = file;

Because of a Pascal language restriction, it is necessary to declare files of type (f_file_type) that are to passed on as parameters to these procedures.

Function F_Open (var fid: F_File_Type):boolean;

This function should be called before any of the following are used. This enables a check to be made on the status of a file. The function returns true if the file is open and false if is not open. The following functions will not give the correct values if the file is not open.

Function F_Length (Var Fid : F_File_Type) : Integer;

Returns the length (in blocks) of the file attached to the Fid identifier. If the file is not opened, the result is returned as zero. This only has meaning for files on blocked devices as the value returned is the number of blocks allocated to the file.

Function F_Unit_number (Var Fid : F_File_Type) : integer;

Returns the unit containing the file attached to the Fid identifier. If there is no file opened to the Fid, the function result is Zero.

Procedure F_Volume (Var Fid : F_File_Type;
Var File_Volume : String);

Returns the name of the volume containing the file attached to the Fid identifier. If the external file lacks a defined volume name, F_Volume returns a volume ID constructed from a unit number (eg. #3:). If there is no file opened to the Fid, the file_volume is set to a null string.

Procedure F_File_Title (Var Fid : F_File_Type;
Var File_Title : String);

Returns the title (with suffix) of the file attached to the Fid identifier. If there is no file opened to Fid, or if the external file is a volume, then the File_title is set to a null string.

Function F_Start (Var Fid : F_File_Type) : integer;

Returns the block number of the first block of the file attached to the Fid identifier. This only has meaning for files on block devices. If there is no file opened to Fid, the function result is returned is zero.

Function F_is_Blocked (Var Fid: F_File_Type) : Boolean;

Returns a boolean that is TRUE if the file attached to the Fid identifier is located on a block-structured unit. If there is no file opened for the Fid or if the device is not block structured , the function result is set to false.

Procedure F_Date (Var Fid : F_File_Type;
Var File_Date : F_Date_Rec);

Returns a record indicating the last access date for the file attached to the Fid identifier. If there is no file opened to Fid, the File_Date is unchanged. The definition of F_Date_Rec type is

```
F_Date_Rec  = Packed Record
                Month : 0..12;
                Day   : 0..31;
                Year  : 0..100;
              End;
```

Unit interface

```
Unit FileInfo;
Interface
Type F_File_Type = file;
F_Date_Rec  = Packed Record
                        Month : 0..12;
                        Day   : 0..31;
                        Year  : 0..100;
                    End;

Function F_Open (var fid:  F_File_Type):boolean;
Function F_Length (Var Fid : F_File_Type) : Integer;
Function F_Unit_number (Var Fid : F_File_Type) : integer;
Procedure F_Volume (Var Fid  : F_File_Type;
                    Var File_Volume : String);
Procedure F_File_Title (Var Fid : F_File_Type;
                    Var File_Title : String);
Function F_Start (Var Fid  : F_File_Type) : integer;
Function F_is_Blocked (Var Fid : F_File_Type) : Boolean;
Procedure F_Date (Var Fid : F_File_Type;
                    Var File_Date  : F_Date_Rec);
```

Example Program:

```
Program File_Demo;

Uses
  (*$UFILE.INFO.CODE*)
  File_Info;

Var Fid : File;
    Name,
    Title,
    Volume : String;
    Start,
    Blocks,
    Unit_Num : Integer;
    Is_Blocked : Boolean;
    Date : F_Date_Rec;

Begin {of File_Demo}
  Write (File Name ? ');
  ReadIn (Name);
  If Length (Name) = 0 Then
    Exit (File_Demo);
```

After a space is typed, the message line will be erased and the cursor will be returned to its previous position when that is possible. If a program uses UNITREADs or UNITWRITEs to the console, the previous cursor position may be lost. Use of GOTOXY (but not SC_GOTOXY) may also cause loss of the previous cursor position. This is because the p-System is not informed of the cursor position after these kinds of low level I/O operations.

II.4.2 User Control of Error Messages

A program may change the line on which an error message is displayed. It may also change the actual message displayed when a code segment is required from a disk that is not present in the appropriate drive for blocked devices. If the code file is on a subsidiary volume, set the message for the principal volume.

The new ERRORHANDLING unit provides these facilities. The file ERRORHAND.CODE on the utilities disk contains this unit. In order to use ERRORHANDLING a Pascal program should have a declaration similar to the following example.

```
{$U errorhand.code} errorhandling;
```

Also, ERRORHANDLING must be available at runtime, either in a library or libraried into the using program's code file. For more information on units and libraries, see The UCSD Pascal Handbook and the Users Manual, Version IV.0.

The following procedures are available within this unit:

```
PROCEDURE SET_ERROR_LINE (LINE:INTEGER)
PROCEDURE SET_USER_MESSAGE (DRIVE:INTEGER; MESG:STRING);
```

A program may change the line on which p-System run-time error messages are to be displayed by calling SET_ERROR_LINE with the desired line number as a parameter LINE. After the call to SET_ERROR_LINE, any run-time error messages will be displayed on that line until SET_ERROR_LINE is used again to specify another line.

The standard message for needed code segments on disks that are not present may be changed by calling SET_USER_MESSAGE with parameter DRIVE set to the physical device number and parameter MESG set to the desired message string.

WARNING: A user message will be destroyed if a MARK is called before a SET_USER_MESSAGE.

NOTE: The physical device numbers are 4, 5, and 9 through the maximum number for physical disk as configured in SETUP.

II.4 Error Messages

Execution error messages are displayed by the p-System under certain circumstances. If a code segment is needed and the disk containing it is not in the appropriate drive, the user is prompted to replace the disk and type <space> to continue. If a program attempts to divide by zero, or access outside the bounds of a Pascal array, a message indicates this and the user is prompted to type <space>, at which point the p-System is re-initialized. These and other execution errors are listed in the Users Manual, Version IV.0.

It is now possible for a program to alter the message that is displayed when certain errors occur. This should be useful for applications developers, especially those whose customers speak languages other than English.

Also, the display format of p-System error messages has been changed in a way that should assist applications developers.

II.4.1 Format of Error Messages

Formerly, error messages were displayed on three lines beginning at the current cursor position. They are now displayed on one specified 80-column line. For example, when a code segment is needed from a disk that is not present in the appropriate drive, the following prompt is displayed.

Need segment SEGNAME: Put volume VOLNAME in unit U then type <space>

This indicates that the segment SEGNAME was not found on volume VOLNAME in device U. By placing the correct volume in the correct unit and typing space, execution should continue normally.

The following example shows the error message that occurs when a user presses the p-System BREAK key.

Program Interrupted by user--Seg PASCALIO P# 17 O# 310 <space> continues

After a space is typed, the p-System will be re-initialized.

System error messages such as these will always appear at a fixed position on the screen. The default position is the bottom line. (Any line can be specified, however.) A BEL character (audible beep) will be written to the console device when the message is written out.

```
  (*$I-*)
  Reset (Fid, Name);
  writeln('Ioresult ',ioresult:6);
  if f_open(fid) then
    Begin
      F_Volume (Fid, Volume);
      Is_Blocked := F_Is_Blocked (Fid);
      Unit_num := F_Unit_number (Fid );
      F_File_Title (Fid, Title);
      F_Date (Fid, Date);
      Blocks := F_Length (Fid);
      Start := F_Start (Fid);
      Writeln;
      Write (Volume, ': (Unit ', Unit_Num);
      If Is_Blocked Then
        Begin
          Writeln (' blocked)');
          Writeln (Title, Blocks:7,
                   Date.Month:4, -Date.Day, -Date.Year, Start:6);
        End {of If Is_Blocked}
      Else
        Writeln (' serial)');
    End {of If F_Volume}
  Else
    Writeln ('No such file');
End {of File_Demo}.
```

Then, if a code segment is required from a missing disk in the unit for which the user program has designated a special error message, that message will be displayed. The p-System will then wait for the user to enter a <space> whether or not the message actually indicates that a <space> is needed. The message line will subsequently be erased, the cursor will be returned to its former position if possible, and execution will continue.

For other kinds of execution errors, a standard p-System message will be displayed on the message line. A fatal error will always cause the p-System to fail. For non-fatal errors, the p-System will wait for the user to enter a <space>. The message line will then be erased, the cursor will return to its former position, and execution will continue (most likely the p-System will re-initialize itself).

To proceed from a non-fatal error, press <esc>.

WARNING: Escaping from a non-fatal error is a dangerous practice since system data may be corrupted.

Error message values set by the user will remain in effect throughout the execution of the program, but will be reset at program termination or whenever p-System re-initialization occurs.

The user program may reset the error handling values to their default values at any time if special output is no longer desired. The missing code segment message can be reset by passing a null string to SET_USER_MESSAGE.

Unknown results may occur on console devices whose screen width is narrower than the message to be displayed.

II.5 Pascal Compiler Enhancements

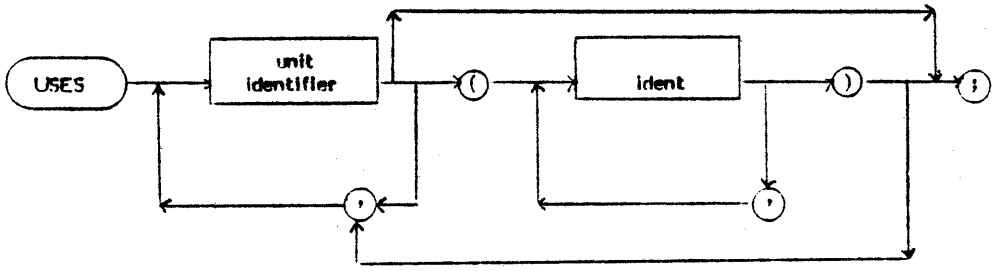
This section describes the new enhancements to the Pascal Compiler and contains examples of their use.

The first section describes an enhancement of the Pascal USES construct. It is now possible to select, from within the client compilation unit, the constants, types, variables, and routines that are needed from the unit. Compile-time symbol table space will not be allocated for the rest of the unit's interface section. If the unit has a large interface section, and the host requires only a small portion of it, a selective USES statement can eliminate the unnecessary symbols and increase the available memory space, thereby increasing the size of a program or unit that can be compiled.

The last section describes enhancements to compiled listings, an additional compiler promptline, and the significance of compiler version numbers in terms of the size of real numbers.

II.5.1 Selective Uses

This enhancement allows users to select certain items from a unit's interface section. The following diagram explains the syntax.



Where ident can be a constant, type, variable, or procedure/process/function, any constant or type used in a selected declaration must be included in the selective USES list. If a selected declaration is not present in the interface text, an error will result during compilation. Most identifiers must be named explicitly in the identifier list if they are to be made available by the USES statement. Some identifiers are available implicitly:

* When an enumerated constant type is explicitly listed, all the constant identifiers of the enumeration are implicitly available.

* When a record type is explicitly listed, all its field names are implicitly available (for example, see the following listing under unit A, line 12, info_rec.)

Finally, list only the name of a routine to make it available. No explicit listing of parameters is needed.

NOTE: The types and constants that these parameters use must be explicitly included.

The first of the following compiled listings is unit A, which is selectively used by units B and C, shown in the second and third compiled listing.

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

Pascal Compiler IV.1 c5s-0      3/24/82            Page 1

=============================================================

    1    2     1:d     1  unit A;
    2    2     1:d     1  interface
    3    2     1:d     1  const
    4    2     1:d     1     maxnum=1000;
    5    2     1:d     1     maxchar=7;
    6    2     1:d     1
    7    2     1:d     1  type
    8    2     1:d     1     byte=0..255;
    9    2     1:d     1     codeblock=packed array
                               [0..maxnum] of byte;
   10    2     1:d     1     alpha=packed array
                               [0..maxchar] of char;
   11    2     1:d     1     ptr_info_rec=^info_rec;
   12    2     1:d     1     info_rec=record
```

```
  13   2     1:d     1                code:codeblock;
  14   2     1:d     1                llink,rlink:ptr_info_rec;
  15   2     1:d     1              end;
  16   2     1:d     1      next=char;
  17   2     1:d     1
  18   2     1:d     1    var
  19   2     1:d     1       first,last:byte;
  20   2     1:d     3
  21   2     1:d     3    function update(var info:ptr_info_rec)
                             :next;
  22   2     1:d     1
  23   2     1:d     1    implementation
  24   2     1:d     1
  25   2     1:d     1      function update;
  26   2     2:0     0      begin
  27   2     2:1     0        with info^ do
  28   2     2:2     3        begin
  29   2     2:3     3          llink:=rlink;
  30   2     2:3    12          if rlink=llink then
  31   2     2:4    22            update:='y'
  32   2     2:3    22          else
  33   2     2:4    27            update:='n';
  34   2     2:2    30        end;
  35   2     1:0     0      end;
  36   2     1:0     0
  37   2      :0     0    end. {unit A}

End of Compilation.

============================================================

Pascal Compiler IV.1 c5s-0      3/24/82              Page 2

============================================================

   1   2     1:d     1    unit B;
   2   2     1:d     1    interface
   3   2     1:d     1    {$U a.code}
   4   2     1:d     1    uses a( {const} maxchar,
                             {include for type ALPHA}
   5   2     1:d     1              {types} alpha,
                             {include for variable WHICH}
   6   2     1:d     1                byte,
                             {include for FIRST and LAST}
   7   2     1:d     1              {vars } first,
                             {include for proc CHANGE}
```

```
 8    2     1:d     1                    last
                        include for proc CHANGE}

                    Using A
 9    2     1:u     1
12    2     1:u     1     maxchar=7;
15    2     1:u     1     byte=0..255;
17    2     1:u     1     alpha=packed array
                            [0..maxchar] of char;
26    2     1:u     1      first,last:byte;
30    2     1:d     3         );
31    2     1:d     1
32    2     1:d     1   procedure change(which:alpha);
33    2     1:d     1
34    2     1:d     1   implementation
35    2     1:d
36    2     1:d           procedure change;
37    2     2:0     0     begin
38    2     2:1     4       if which='          ' then
39    2     2:2    14         last:=first
40    2     2:1    14       else
41    2     2:2    26         first:=last;
42    2     1:0     0     end;
43    2     1:0     0
44    2      :0     0   end; {unit B}
45    2      :0     0
46    2      :0     0
47    2     1:0     0   unit C;
48    2     1:0     0   interface
49    2     1:0     0   implementation
50    2     1:0     0   {$U a.code}
51    2     1:0     0   uses a( {const} maxnum,
                          {include for type CODEBLOCK}
52    2     1:0     0                   maxchar,
                          {include for type ALPHA}
53    2     1:0     0                   byte,
                          {include for type CODEBLOCK}
54    2     1:0     0           {type}  alpha,
                          {include for variable MINE}
55    2     1:0     0                   info_rec,
                          {include for PTR_INFO_REC}
56    2     1:0     0                   ptr_info_rec,
                          {include for func_UPDATE}
57    2     1:0     0                   codeblock,
                          {include for INFO_REC}
58    2     1:0     0                   next,
```

```
                                {include for func UPDATE}
                             Using A
   59    2     1:0     0
   61    2     1:u     1      maxnum=1000;
   62    2     1:u     1      maxchar=7;
   65    2     1:u     1      byte=0..255;
   66    2     1:u     1      codeblock=packed array
                                [0..maxnum] of byte;
   67    2     1:u     1      alpha=packed array
                                [0..maxchar] of char;
   68    2     1:u     1      ptr_info_rec=^info_rec;
   69    2     1:u     1      info_rec=record
   70    2     1:u     1             code:codeblock;
   71    2     1:u     1             llink,rlink:ptr_info_rec;
   72    2     1:u     1               end;

============================================================

Pascal Compiler IV.1 c5s-0  -  a.code      3/24/82  Page 3

============================================================

   73    2     1:u     1       next=char;
   78    2     1:u     1    function update
                               (var info:ptr_info_rec):next;
   79    2     1:u     3
   80    2     1:d     3             {func}  update),
                             Using B
   81    2     1:d                   b;
   82    2     1:d
   83    2     1:d           var
   84    2     1:d              info:ptr_info_rec;
   85    2     1:d              mine:alpha;
   86    2     1:d     0
   87    2     1:0     0    begin
   88    2     1:1     0      new(info);
   89    2     1:1     7      new(info^.rlink);
   90    2     1:1    16      info^.llink:=nil;
   91    2     1:1    22      mine:='newsystm';
   92    2     1:1    30      if update(info)='y' then
   93    2     1:2    39        writeln('info updated')
   94    2     1:1    59      else
   95    2     1:2    61        change(mine);
   96    2      :0     0    end.

End of Compilation.
```

The selective USES list feature will save space in memory during compilation time, and it is good documentation. That is, it identifies all the data declarations that are used for each unit. See chapter III for more information.

All programs that use a special modified version of the system KERNEL or any other unit that has been tailored in-line for a certain program should take advantage of selective USES.

II.5.2 Enhancements to the Compiled Listing

The following items describe enhancements to the compiled listing.

1. The following prompt allows the user to name a listing file.

   Output file for compiled listing? (<ret> for none)

   An 'L' compiler directives can override the file specified. If the user ommits .text from the file name, the compiler will append .text to the end of the file name. To exit the compiler, press <esc>.

2. The system date is now in the heading of a compiled listing.

3. The additional line 'USING <UNITNAME>' will make interface sections from a USED unit easily identifiable on a listing. Also, a 'u' in the line heading (previously a 'd') signifies that the following line is an interface text, not text in the compiled program.

4. Only the declarations chosen from a unit will be present in a compiled listing. If one variable is chosen from a list of variables, the entire list will be present in the compiled listing. For example, if an 'x' is chosen out of the following list,

   ```
   x,y,z,a,b,c,
   d,e,f:boolean;
   ```

5. The compiled listing will include y,z,a,b,c,d,e, and f.

6. The Pascal Compiler version number is comprised of the version number and the real-size default. For example, Pascal Compiler Version c5s-2, means version c5, symbolic degugging is available, and the real-size default is two words. Use the following directives to override the default.

   {$R2} and {$R4}

II.5.3 Pascal Compiler Back-End Errors

Compiler back-end errors can result from a variety of problems. Basically they occur when the back end finds itself or the intermediate code file in an unexpected state. (The intermediate code file is a file used by the compiler to communicate between the front end and back end of the compiler. It consists of compiler directives intermixed with actual P-code.) Back-end errors can be caused by a corrupt intermediate code file, external forces (e.g. bad blocks in the compiler), or source file information that is skipped by the front end but used by the back end.

The following table is a list of back-end errors and a possible explanation for the error.

Error Number	Comments
-1	While trying to generate the constant pool information for a particular code segment, the back end tries to read one block from the intermediate code file and the read fails.
1	If the lexical procedure nesting is greater than 31, this error will occur. Since the front end only allows nesting of seven procedures, this error should theoretically never occur.
4	The intermediate code file directives are bytes with values greater than 252. If the back end reads a directive with a value that is less than 253, error number 4 will result.
5	The current procedure number is greater than the maximum number of procedures for that segment.
6	The operator (variable,constant,jump location) that the back end is trying to remap is not in the scope of the compilation unit.
7	The back end can not find the target site to jump to while resolving jumps.
8	There are more than 400 jumps in the jump table while trying to enter a site jump error. Try dividing each procedure with many jumps into more than one procedure.
9	There are more than 400 jumps in the jump table while trying to enter a target jump. Try dividing each procedure with many jumps into more than one procedure.
11	The code pointer is less than 0 or greater than the length of the intermediate code file while building a jump table.
12	A jump site can not be found in the jump table.

Error Number	Comments
22	Unexpected end of input while generating the LCO P-code instruction.
23	Unexpected end of input while generating the LDC P-code instruction.
24	The exit for a certain procedure can not be found in the jump table.
25	The code pointer is less than 0 or greater than the length of the intermediate code file while generating p-code.
27	The code pointer is less than 0 before trying to read in more code from the intermediate code file to the code buffer.
28	The codepointer is less than 0 after trying to read in more code from the intermediate code file to the code buffer.
29	The current final output block number is greater than the block number of the intermediate code file being processed.
30	The final code file size exceeds the intermediate code file size before trying to write more final code.
31	The final code file size exceeds the intermediate code file size after writing more final code.
41	The line length of a compiled listing exceeds 120 characters. (Note: this error can occur on a pre-IV.1 compiler if there is an illegal character after a DLE character.)
86	Could not find a particular segment in the intermediate code file.
99	The number of procedures does not match the number specified in the procedure dictionary.

Recovery Options:

If a syntax error has occurred in the front end and a back-end error occurs, fix the syntax error and try recompiling.

If there are bad blocks on any of the disks being used for the compilation replace the bad disks with good ones and try recompiling.

If the bug persists and you purchased a fully adaptable system, please report the problem to the p-System supplier.

II.6 Extended Memory

Extended memory is a new feature that allows the current release of the p-System to run in environments of up to 128K of memory. This is accomplished by dividing the p-System run time environment into two parts, each of which may occupy as much as 64K bytes of memory.

The code pool is an area of memory where most code segments are executed by the p-System. This code includes the operating system, filer, editor, etc., as well as user programs. In non-extended memory systems, the code pool shares the same space with the rest of the p-System (i.e. the interpreter, RSP, BIOS, SBIOS, and the p-System stack and heap). The code pool resides between the stack and heap on non-extended memory systems.

On extended memory systems, the code pool is placed in a separate area of memory all together. Thus the code pool may occupy an entire 64K portion of RAM, and the rest of the p-System may occupy another entire 64K area.

A major advantage of the extended memory feature is the additional memory space available for executable code to use. This means that larger programs can be compiled and executed.

Also, the code segments on extended memory systems may not need to be moved or swapped as often as those on non-extended memory systems thereby producing significant performance improvements.

Because there is more space for the p-System stack and heap to grow, the chances of a stack overflow are reduced.

II.7 Echoed Characters

The user can now designate the character codes that the p-System will echo to the screen. Previously, the system automatically echoed ASCII codes 13 and 32 through 126. Now, the user may select any subset of the character codes from 0 through 255 (see SETUP in Chapter IV of this supplement under printable characters).

II.8 Editor Enhancements

II.8.1 Setting Tab Stops

The editor will now allow the user to set tab stops. From the E(dit command prompt, press S(et, E(nvironment, and then press S(et tabstops. The system will display the following interface prompt.

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

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

Column#1
```

The cursor will start at position one in the line of Ts and dashes (-). The line 'Column#1' indicates the positon of the cursor. To set or remove a tab, move the cursor to the desired location using the right or left vector keys, or press 'C' and enter the desired column number. Press 'T' to insert a tab or delete a tab. Pressing 'T' will change the indicator from a dash to T; pressing 'T' again in the same column will change the 'T' back to a dash. The system displays the current column number of the current cursor position and updates it each time a right/left vector key or 'C(olumn' command is pressed.

The Version IV.1 Editor is not compatible with existing IV.0 Operating Systems. It uses either standard or ANSI SCREENOPS.

II.8.2 Editing Line in S(et E(nvironment Display

The following listing is an example of the S(et E(nvironment display showing a new line above the date of file creation line. The new line identifies the file currently being edited.

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

    3152 bytes used, 29612 available.


Editing:  SCHEDULE.TEXT
Created March 10, 1982; last updated March 24, 1982 (revision
10)
Editor Version .
```

If the file has just been created but not named, the line will read--

```
Editing:  unnamed
```

II.8.3 Byte-Sex Independent Environment Information

The environment information in the first two blocks of a text file is no longer byte-sex dependent. That is, the user can edit a file on a machine that has a different byte sex than the machine on which the file was created, and the environment information will be available.

II.8.4 Editor Control Characters

The restrictions on the values for the following data fields in SYSTEM.MISCINFO have been removed.

```
EDITOR ACCEPT KEY
EDITOR ESCAPE KEY
KEY TO DELETE CHAR
KEY TO DELETE LINE
KEY TO MV CURS DOWN
KEY TO MV CURS LEFT
KEY TO MV CURS RGHT
KEY TO MV CURS UP
```

The user can now use SETUP to designate which key or keys on the terminal will control the corresponding editing function.

For example, to set up a function key whose value is <esc> A, set up the Editor Escape Key to be prefixed A.

If any functon key has the same value as the keyboard prefix, then the user must indicate in SETUP that the key is prefixed and must press the key twice to make it function.

II.9 The 8086 and 68000 Assemblers

This section contains information that is specific to the 8086 and 68000 versions of the UCSD p-System Adaptable Assembler.

II.9.1 The 8086 Assembler

The UCSD p-System 8086/88/87 Assembler differs in some respects from the standard Intel Assembler. These differences are listed in this section. Also, the 8086/88/87 specific assembler errors are listed.

II.9.1.1 Notational Conventions

The following paragraphs define the 8086 notational conventions.

Assembler Directives. None of the Intel Assembler directives or operators are implemented. Instead, the assembler directives described in the Users Manual, Version IV.0 are available.

Parentheses. Index or base register references in a memory operand are enclosed in parentheses, not square brackets, e.g., FIRST(BX) rather than FIRST[BX].

Immediate Byte. The following example shows how the ADD immediate byte to memory operand is coded to distinguish it from the ADD memop, immedword, which is the default.

```
ADDBIM   memop,immedbyte
```

Similarly, MOVBIM, ADCBIM, SUBBIM, SBBBIM, CMPBIM, ANDBIM, ORBIM, XORBIM, and TESTBIM are added to the vocabulary.

Memory Byte. The following example shows how an INC memory byte is coded to distinguish it from INC memory word, which is the default.

```
INCMB    memop
```

Similarly, DECMB, MULMB, IMULMB, DIVMB, IDIVMB, NOTMB, NEGMB, ROLMB, RORMB, RCLMB, RCRMB, SALMB, SHLMB, SHRMB, SARMB were added to the vocabulary to specify memory byte operands.

MUL and DIV Byte. In MUL, IMUL, DIV, IDIV the single memory operand form implies a word operation. For example,

```
MUL      memop
```

To specify a byte operation, either MULMB memop or the following form may be used.

```
MUL     AL,memop
```

The same is true for IMUL, DIV, IDIV.

NOTE: The DIV AL,memop is rather misleading because the actual operation would be AX/memory-byte.)

MOV Substitute for LEA. For LEA reg,label or LEA reg,label+const the assembler will substitute MOV reg,immed_val for immed_val = label or label+const. This saves four clock times (4 vs. 8).

IN and OUT. The normal form of IN and OUT is IN ac,port or IN ac,DX and OUT port,ac or OUT DX,ac where ac=AL denotes an 8-bit data path and ac=AX denotes a 16-bit path. Since the accumulator is the only possible register source/destination (DX specifies port=address in DX), single operand forms are also provided: INB and OUTB for byte data, and INW and OUTW for 16-bit data. The syntax is INB port or INB DX.

In the two-operand forms of IN and OUT, the order of the operands is not important; thus OUT ac,DX or OUT ac,port will be acceptable.

String Operations. The mnemonics for the string operations are suffixed with B or W to denote byte or word operations: thus MOVSB and MOVSW, CMPSB and CMPSW, SCASB and SCASW, LODSB and LODSW, and STOSB and STOSW are in the vocabulary, but MOVS ... STOS are not.

Segment Override. XLAT and the string instructions have implied memory operands and no code is required in the operand field. However, in order to permit the specification of a segment override prefix in the case of XLAT, MOVSB/MOVSW, CMPSB/CMPSW, and LODSB/LODSW, the assembler permits operand expressions for these instructions.

NOTE: The only the default segment for SI, namely DS, can be overridden. The segment for DI is ES and cannot be overridden. A segment override prefix of DS applied to SI does not generate a segment override prefix.

If these operations were written with operands, they would have the following syntax.

```
XLAT          AL,(BX)
MOVS{B|W}     (DI),[seg:](SI)
CMPS{B|W}     (DI),[seg:](SI)
SCAS{B|W}     (DI),AX
LODS{B|W}     AX,[seg:](SI)
STOS{B|W}     (DI),AX
```

The string instructions may be prefixed by a REP (repeat) instruction of some type. The assembler flags an error if both REP and a segment override are specified.

In addition to the forms DS:memop, etc., a separate mnemonic SEG followed by a segment register name may be written in a statement preceding the instruction mnemonic.

Examples:

```
MOV   AX,ES:AVALUE
```

is equivalent to

```
SEG   ES   MOV   AX,AVALUE
```

Long Jumps, Calls, and Returns. Intersegment CALL, RET, and JMP are implemented as follows:

a. The mnemonics CALLL, RETL, and JMPL specifically designate intersegment operations.

b. An indirect address (e.g., (reg) or (label)) is assembled in standard fashion with a "mod op r/m" effective address byte possibly followed by displacement bytes. The memory location referenced must hold the new IP, and the next higher location must hold the new CS.

c. The direct address form must have two absolute operands:

```
CALLL    expr1,expr2
```

where expr1 is the new IP and expr2 becomes the new CS. Constants or external symbols (e.g., .REF definitions) qualify as absolute operands.

8087 Mnemonics. Mnemonics for the 8087 floating point operations are standard except for certain of the memory reference operations, where a letter suffix is appended to denote the operand size:

__D short real or short integer (double word)

__Q long real or long integer (quad word)

__W integer word

__T temporary real (ten byte)

The D and Q suffixes apply to the following real ops:

FADD, FCOM, FCOMP, FDIV, FDIVR, FMUL, FST, FSUB, FSUBR, FLD, FSTP

e.g., FADDD, FADDQ, etc.

The T suffix applies only to FLD and FSTP.

The W and D suffixes apply to the following integer ops:

FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FIST, FISUB, FISUBR, FILD, FISTP

The Q suffix for long integers applies only to FILD and FISTP.

II.9.1.2 8086/88/87 Assembler Error Messages

The following error messages are specific to the 8086/88/87 Assembler:

76 : Had label, open parenthesis then illegality
77 : Expected absolute expression
78 : Both operands connot be a segment register
79 : Illegal pair of index registers
80 : Have to use BX,BP,SI or DI
81 : Illegal constant as first operand
82 : The first operand is needed
83 : The second operand is needed
84 : Expected comma before 2nd operand
85 : Registers stand alone except in indirect
86 : Only 2 registers per operand
87 : Expected label or absolute
88 : Illegal to use BP indirect alone
89 : Close parenthesis expected

90 : Cannot POP CS
91 : Cannot have exchange of r8 with r16
92 : Segment registers not allowed
93 : ESC external op on left must be const < 64
94 : Only one of the operands can have segment override
95 : Right operand must be a memory location
96 : Left operand must be a 16-bit register
97 : Left operand must be memory or register alone
98 : Op cannot be a segment register or immediate
99 : Count must be 1 or in CL
100 : A byte constant operand is required
101 : Operand must use () or be a label
102 : LOCK followed by something illegal
103 : REP precedes only string operations
104 : Not implemented
105 : Expected a label
106 :
107 : Open parenthesis expected
108 : Expected register alone as right operand
109 : Segovpre then regalone, thats illegal
110 : Only one operand allowed
111 : Operands are AL,op2 for byte MUL, etc.
112 : SP can only be used with the SS segment
113 : MOVBIM only for immediate to memory
114 : BIMs must be immediate bytes to memory
115 : Seg override on repeated instruction not ok
116 : Segment register expected
117 : (8087) Invalid two-operand format
118 : (8087) Invalid single operand format
119 : (8087) Improper operand field
120 : (8087) Instruction has no operands
121 : No override of ES on string destination
122 : Interseg needs 2 constant or external operands
123 : I/O port must be immediate byte or DX
124 : I/O source/dest register must be AL or AX

II.9.1.3 Sharing Machine Resources with the Interpreter

II.9.1.3.1 Calling and Returning

The p-System Interpreter invokes an assembly routine using the call long (CALLL) operator. Thus, the top of the stack contains a two-word return address when the operating system enters the routine. To return from an assembly routine, use the return long (RETL) operator, or pop the return address and use a jump long (JMPL) operation.

II.9.1.3.2 Accessing Parameters

To access parameters that are passed to an assembly routine, move the value of register SP (which points to the P-machine stack) into BP. Access the parameters by adding an offset of 4 bytes (to account for the two-word return address. The first parameter (at the location 4 bytes above the top of the stack) is actually the last declared parameter in the host routine (the parameters are pushed in the order that they are declared).

If a .FUNC assembly routine is to return a function value, place the function value just above the last parameter using the same accessing scheme. The size of the returned function value is either 1, 2 or 4 words.

Give the RETL operator an operand that indicates how many bytes to cut the stack back after popping its two-word return address. Use the size of the data space occupied by the parameters. Thus, the user may access parameters and make a clean return without using a specific POP or PUSH instruction.

The following listing is an example of the scheme of accessing parameters and returning.

```
MOV   BP,SP
MOV   AX,(BP+4) ;Last Param
MOV   BX,(BP+6) ;Middle Param
MOV   CX,(BP+8) ;First Param
                .
                .
                .

MOV   (BP+10),RSLT ;Function return val
                   ; (if .FUNC)

RETL  6            ;Remove 3 params
```

II.9.1.3.3 Register Usage

All of the 8086/88 registers are available for use by user assembly routines (the interpreter saves and restores the register values that it needs).

SS and SP must be preserved by the user, however. (The user may create and use a private stack if a minimum of 40 words are left available for stack expansion during interrupts. This is a very dangerous procedure, however, and is not recommended.)

NOTE: The integrity of the P-machine stack must be maintained. This is the programmer's responsibility and if this is not done, the results are unpredictable.

When the operating system enters the assembly routine, SS equals the P-machine stack pointer (SP). Also, DS, ES, and CS are all equal to the base of the p-System code segment.

Parameters that are passed as Pascal VAR variables are p-System pointers to actual data. These pointers are relative to SS.

.PRIVATE and .PUBLIC variables are are also SS relative.

.BYTE quantities, .WORD quantities, and .REF'f labels are relative to CS, DS, or ES.

II.9.2 68000 Assembler Syntax Conventions

The 68000 Assembler follows Motorola standard syntax for opcode fields, register names and address modes. The following list points out some restrictions of the p-System Adaptable Assembler.

Only the absolute short address mode is available. The absolute long address cannot be generated by the assembler.

Labels may not be accessed with the absolute address mode.

References to labels with a .PROC or .FUNC generate the P(-relative address mode.

An external label may only be accessed as a displacement from an address register.

Immediates above FFFFH cannot be generated.

Opcodes which have an optional suffix of A,I,M,Q or X must contain

that suffix explicitly.

Length qualifiers (.B, .W or .L) must be specified explicitly in those instructions which have a choice of length. All other instructions must not contain a length qualifier.

The following instructions must contain a length qualifier:

ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL (register), ASR (register), CLR, CMP, CMPA, CMPI, CMPM, EOR, EORI, EXT, LSL (register), LSR (register), MOVE (except special forms), MOVEA, MOVEM, MOVEP, NEG, NEGX, NOT, OR, ORI, ROL (register), ROR (register), ROXL (register), ROXR (register), SUB, SUBA, SUBI, SUBQ, SUBX, TST

The following instructions must not contain a length qualifier:

ABCD, ASL (memory), ASR (memory), BCHG, BCLR, BSET, BTST, CHK, DBcc, DIVS, DIVU, EXG, JMP, JSR, LEA, LINK, LSL (memory), LSR (memory), MOVE to CCR, MOVE to SR, MOVE from SR, MOVE USP, MOVEQ, MULS, MULU, NBCD, NOP, PEA, RESET, ROL (memory), ROR (memory), ROXL (memory) ROXR (memory), RTE, RSR, RTS, SBCD, Scc, STOP, SWAP, TAS, TRAP, TRAPV, UNLK

The following instructions may contain an optional length qualifier of .S (generate short forward branch):

Bcc, BRA, BSR

II.9.2.1 Miscellaneous

The 68000 processor is byte-addressed and word-oriented. The byte sex is most-significant byte first.

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

II.9.2.2 68000 Error Messages

The following error messages are specific to the 68000 Assembler:

76: unrecognizable address mode
77: address register expected
78: close paren ')' expected
79: displacement out of range
80: index register expected
81: illegal length qualifier
82: illegal source address mode

83: illegal destination address mode
84: comma ',' expected
85: length qualifier required
86: length qualifier not allowed
87: data register expected
88: label expected
89: illegal register list
90: immediate operand expected

II.10 Turnkey Applications Facilities

II.10.1 SYSTEM.MENU

If an applications programmer wishes to design display prompts for a particular application, he or she may write a program called *SYSTEM.MENU that will display a menu for his or her particular applications environment. The main p-System prompt line need never be displayed if the program chaining facilities are used (see the Users Manual, Version IV.0).

The user must place the compiled code file called SYSTEM.MENU on the system disk. The p-System will execute SYSTEM.MENU each time the system prompt line would normally appear.

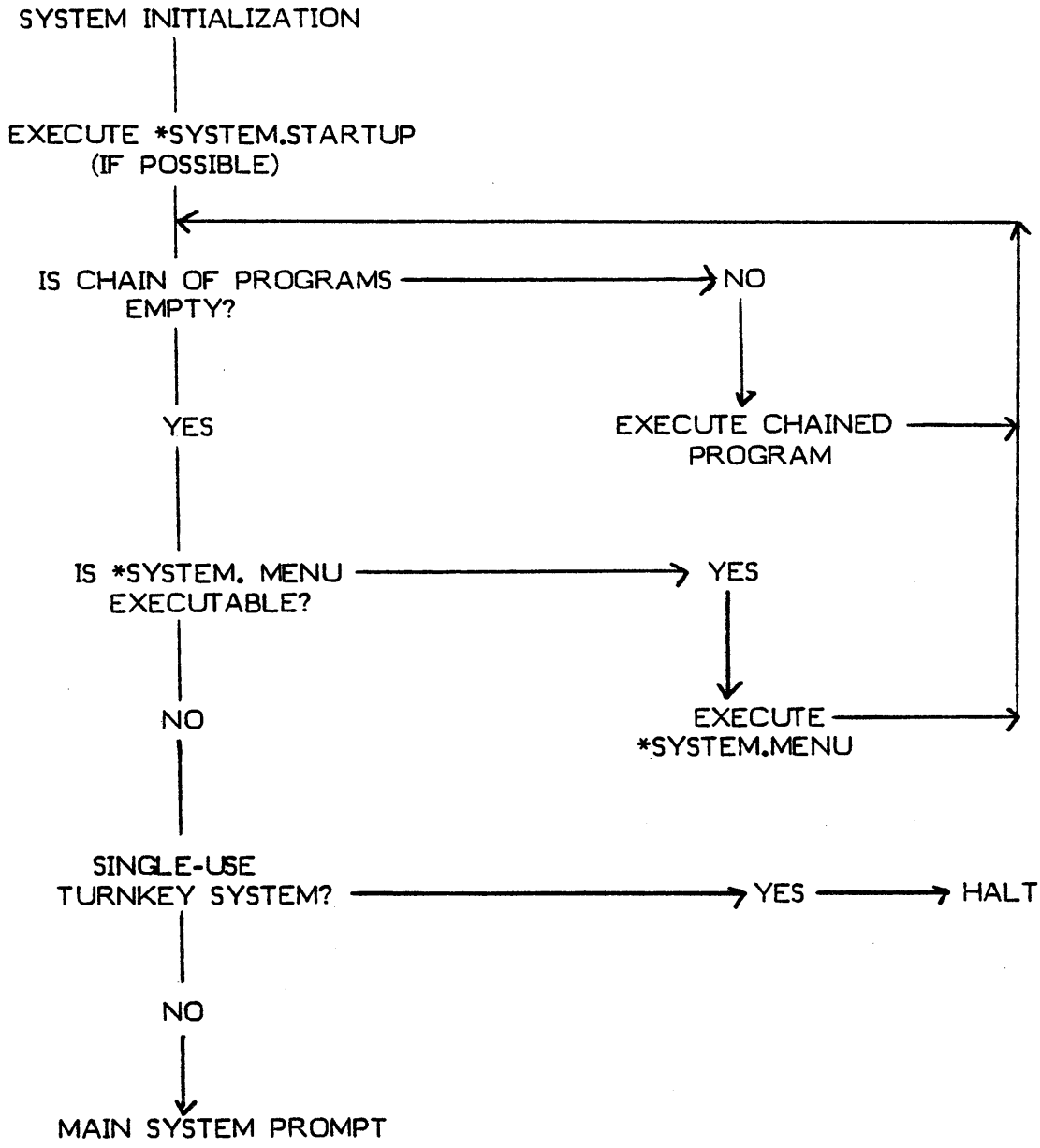
SYSTEM.MENU works much like SYSTEM.STARTUP. If it is present, the system will execute SYSTEM.MENU any time the user initializes the system.

II.10.2 Single-Use Turnkey System

Single-use turnkey is a version of the p-System designed to package and execute a single application program or series of related programs called via the p-System chain mechanism. The package may contain SYSTEM.STARTUP, but must contain an executable code file called SYSTEM.MENU. The facilities on the main system prompt line such as the editor and the filer are not available.

II.10.3 System Initialization

The following diagram illustrates the implementation and the flow of control within the operating system.



II.11 Performance Monitor

The user can gain access to performance information by writing a unit called PERFOPS and including it in the operating system. The p-System contains the following interface for PERFOPS.

```
UNIT PERFOPS:
  USES KERNEL;
  PROCEDURE PM_Fault;
  PROCEDURE PM_Dump_Seg (SegToDump : SIB_P);
  PROCEDURE PM_Prog_Begin;
  PROCEDURE PM_Prog_End;
  PROCEDURE PM_Start_Stop (Start : BOOLEAN);
  PROCEDURE PM_Interactive;
IMPLEMENTATION
```

With the exception of PM_Start_Stop and PM_Interactive, the operating system calls these procedures to indicate actions that the system is taking. Calls to these procedures by the operating system will only be made if the boolean, Has_PM, located in the KERNEL interface section, is set to true.

The following paragraphs describe the procedures.

PM_Fault. The operating system calls this procedure each time it enters the fault handler. (A fault occurs whenever a code segment is needed from disk, when the stack is about to run into the code pool or the heap, when the heap is about to run into the code pool or the stack, or if a pool fault occurs on systems with extended memory.) This procedure must not cause an additional fault; it must not call any procedure that may not be in memory. Stack space requirements must be minimized.

PM_Dump_Seg. The operating systems calls this procedure from the fault handler. PM_Dump_Seg indicates that SegToDump is being removed from the code pool. This procedure must not cause an additional fault; it must not call any procedure that may not be in memory. Stack space requirements must be minimized.

PM_Prog_Begin. The operating system calls this procedure to indicate that a program is about to start.

PM_Prog_End. The operating system calls this procedure to mark the end of a program.

PM_Start_Stop. This entry point controls performance monitoring. This procedure will memlock PERFOPS and set Has_PM to true, or vice versa depending on the value of parameter start. The user must call this routine before PERFOPS can be used. It should be called again to deactivate PERFOPS.

NOTE: PERFOPS must be memlocked before setting Has_PM to true.

PM_Interactive. The 'I' command in the debugger will call this procedure if Has_PM is true. This routine will provide data gathered by the first four procedures of PERFOPS. In this way, a user can use PERFOPS interactively from the debugger.

II.12 REALCONV Utility

REALCONV is a utility that converts real constants in a IV.0 code file from canonical (compiled) form to native machine format. It eliminates the need to convert real constants at segment load time, thus increasing the initial loading speed of the program segments, as well as the overall run-time speed of the program. This is especially important for programs that require frequent loading of segments containing real constants.

The real constant conversion utility is a filter that works on code files, replacing canonical reals by run-time reals in-place. Hence, when the source file is not available, it is advisable to make a backup copy of the code file to be processed before executing the utility program. This will avoid destroying the code file while executing REALCONV (with an unsuccessful block write).

Because the conversion algorithm uses real arithmetic of the host processor, the utility must be executed on the processor on which the output file will run. In most cases, a code file produced by the utility will not run on another processor, which reduces the portability of otherwise completely transportable code. The user may choose to sacrifice portability for execution speed on a time-critical program.

To use the REALCONV, press X(ecute at the command level. The system will respond with the following prompt.

```
EXECUTE WHAT FILE?
```

Enter REALCONV, then press <return>. The utility program will display the following prompt.

```
ENTER FILE NAME:
```

Enter the name of the code file to be processed, and then press <return>. It is not necessary to include the suffix .code.

If REALCONV cannot complete the conversion successfully, it will print a message and stop. The following listing shows examples of possible messages.

```
NOT ENOUGH MEMORY
ERROR IN READING...(The dots represent segment dictionaries,
      first block, constant pool or segment as the case may be)
ERROR IN WRITING SEGMENT
TOO MANY DICTIONARIES
```

The following paragraphs describe the error messages.

NOT ENOUGH MEMORY. The segment to be processed is larger than the available memory space.

ERROR IN READING. Execute REALCONV again.

TOO MANY DICTIONARIES. There are more than 80 segments in the file.

If REALCONV executes successfully, the system displays a dot on the console for each segment converted, and once the conversion is completed, the system displays the following message.

ENTER FILE NAME:

The user may enter the name of another file for processing or press <return> to exit REALCONV and return to the command level prompt line.

III NEW OPTIONAL FACILITIES

III.1 Native Code Generator

Native code generators (NCG) are utility programs that translate selected portions of an executable P-code file into native code (n-code). Using native code directives inserted into the source code, the user indicates which portions of the file are to be translated. The result of this procedure is an equivalent p-System code file that contains P-code and n-code. The NCG will translate only valid executable code files produced by a UCSD p-System compiler.

Because n-code generally executes faster than P-code, the NCG can be used to speed up the execution of selected portions of P-code, for example, portions of code where most of the run time is spent. However, P-code was designed for compactness and consequently takes up less space in memory than n-code. To use the NCG effectively, translate only those portions of P-code for which execution time is critical. Misuse of the NCG can greatly increase the size of the code file.

The user indicates that portions of code are to be translated by inserting native code directives into the source file before compilation. The following compile-time switches are the native code directives.

$N+ and $N-

The user inserts the first switch {$N+} where the translation should begin and inserts the last switch {$N-} where the translation should end. When the compiler encounters the first switch, it begins generating the additional P-code necessary for n-code generation and stops generating when it encounters the last switch. The default setting for this compiler option is {$N-}. (This notation applies to USSD Pascal. Similar notations apply to other languages.)

III.1.1 Native Code Directives and Pascal

Because the NCG translates a Pascal code file on a procedure by procedure basis, only a complete procedure (function or process as defined in UCSD Pascal) can be translated. One set of native code directives may designate more than one procedure; but the native code generation cannot begin within the body of a procedure. The following example shows the use of the native code directives in Pascal.

```
function MAX (a,b:  integer): integer;
  {$N+}
  begin
    if a > b then MAX:= a else MAX:= b;
  end;
  {$N-}
```

The object code file, produced by the compiler from source code containing native code directives, is an executable P-code file that maintains its machine portability. The only difference is that the native code directives slightly increase the size of the object code file.

III.1.2 Native Code Directives and BASIC

The native code directives ($N+ and $N-) can be inserted into the BASIC source code file at any point within a procedure. The user can specify translation on a statement by statement basis. The following example shows the use of native code directives in BASIC.

```
I=3
GOSUB 100
I=4
GOSUB 100
STOP
{$N+}
100 REM   THE SUBROUTINE
FOR J=1 TO 100
PRINT I^5,I^.5
{...}
{$N-}
{...}
NEXT J
RETURN
END
```

III.1.3 Native Code Directives and FORTRAN

To designate code for translation in a FORTRAN source code file, the user must place the native code directive {$NATIVE} before the first statement function or executable statement in a procedure. The native code directive must begin in the first column of the line. The translation directive still applies for the entire procedure. The following example shows the use of native code directives in FORTRAN.

```
      FUNCTION MAX (I,J)
$NATIVE
      MAX=I
      IF (J .GT. I) MAX=J
      RETURN
      END
```

III.1.4 Running the Native Code Generator (NCG)

The NCG is run by executing one of the following files, depending on the processor.

* Z80.NCG.CODE
* 8080.NCG.CODE
* 8086.NCG.CODE
* 9900.NCG.CODE

The NCG generates a prompt asking the user for an input code file and an output code file. The output file must contain the suffix .CODE . Only executable code files can be translated by the NCG (they must be already linked).

The NCG will produce a formatted listing of the code generated for each procedure it translates. The NCG generates a prompt asking the user for the name of a listing file. To produce a listing, enter a listing file name, for example, Console:, Printer:, #5:List, List.Text. To eliminate the listing, press the return key in response to the prompt.

The following listing is an example of function MAX translated on the Z80 NCG.

```
============================================================
Final Z80 Code for segment TEST     procedure 2
Segment offset 30
Source Object                                  .RADIX  10
P-Code N-Code
(Dec. Offsets)                    MP           .EQU    BC
============================================================

            0|                                 .WORD   91,-30
            0|
     4:     0| A8                ;P-code      NATIVE
            1| DD5E0A                         LD      E,(IX+10)
            4| DD560B                         LD      D,(IX+11)
            7| DD6E0C                         LD      L,(IX+12)
           10| DD660D                         LD      H,(IX+13)
           13| 7A                             LD      A,D
           14| AC                             XOR     H
           15| F23400                         JP      P,L1
           18| A2                             AND     D
           19| C33800                         JP      L2
           22| 7B                L1:          LD      A,E
           23| 95                             SUB     L
           24| 7A                             LD      A,D
           25| 9C                             SBC     H
           26| F24B00            L2:          JP      P,L3
     9:    29| 210E00                         LD      HL,14
           32| 09                             ADD     HL,BC
           33| DD5E0C                         LD      E,(IX+12)
           36| DD560D                         LD      D,(IX+13)
           39| 73                             LD      (HL),E
           40| 2C                             INC     L
           41| 72                             LD      (HL),D
    11:    42| C35800                         JP      L4
    13:    45| 210E00            L3:          LD      HL,14
           48| 09                             ADD     HL,BC
           49| DD5E0A                         LD      E,(IX+10)
           52| DD560B                         LD      D,(IX+11)
            5| 73                             LD      (HL),E
            6| 2C                             INC     L
           57| 72                             LD      (HL),D
    15:    58| CD4200            L4:          CALL    INTRP_REL+66
    15:    61|                   ;exit code
           61| 9602              ;P-code      RPU     2
```

The following listing is an example of function MAX translated on the 8086.

```
============================================================
Final 8086 Code for segment TEST  procedure 2
Segment byte offset 30
Source Object                              .RADIX  10
P-Code n-Code                    MP        .EQU    BP
(Dec. Offsets)                   BASE      .EQU    DX
============================================================

           0|                                .WORD   34,0
           0|
    4:     0| A8                ;p-code  NATIVE
           1| 33C0                       XOR     AX,AX
           3| 8B5E04                     MOV     BX,4[BP]
           6| 3B5E02                     CMP     BX,2[BP]
           9| 7F01                       JG      L1
          11| 40                         INC     AX
          12| D1E8              L1:      SHR     AX,1
          14| 7208                       JC      L2
    9:    16| 8B4604                     MOV     AX,4[BP]
          19| 894606                     MOV     6[BP],AX
   11:    22| EB06                       JMP     L3
   13:    24| 8B4602            L2:      MOV     AX,2[BP]
          27| 894606                     MOV     6[BP],AX
   15:    30| FF1E0400          L3:      CALL    4
   15:    34|                   ;exit code
          34| 9602              ;p-code  RPU     2
```

The following listing is an example of function MAX translated on the 8080.

```
============================================================
Final 8080 Code for segment TEST   procedure 2
Segment offset 30
Source Object                                  .RADIX  10
P-Code N-Code
(Dec. Offsets)                    MP           .EQU    BC
============================================================

            0|                                  .WORD   96,-30
            0|
    4:      0| A8                ;p-code       NATIVE
            1| 210A00                          LD      HL,10
            4| 09                              ADD     HL,BC
            5| 5E                              LD      E,(HL)
            6| 2C                              INC     L
            7| 56                              LD      D,(HL)
            8| 210C00                          LD      HL,12
           11| 09                              ADD     HL,BC
           12| 7E                              LD      A,(HL)
           13| 2C                              INC     L
           14| 66                              LD      H,(HL)
           15| 6F                              LD      L,A
           16| 7A                              LD      A,D
           17| AC                              XOR     H
           18| F23700                          JP      P,L1
           21| A2                              AND     D
           22| C33B00                          JP      L2
           25| 7B                L1:           LD      A,E
           26| 95                              SUB     L
           27| 7A                              LD      A,D
           28| 9C                              SBC     H
           29| F24F00            L2:           JP      P,L3
    9:     32| 210C00                          LD      HL,12
           35| 09                              ADD     HL,BC
           36| 5E                              LD      E,(HL)
           37| 2C                              INC     L
           38| 56                              LD      D,(HL)
           39| 210E00                          LD      HL,14
           42| 09                              ADD     HL,BC
           43| 73                              LD      (HL),E
           44| 2C                              INC     L
           45| 72                              LD      (HL),D
   11:     46| C35D00                          JP      L4
```

```
13:     49| 210A00              L3:     LD      HL,10
        52| 09                          ADD     HL,BC
        53| 5E                          LD      E,(HL)
        54| 2C                          INC     L
        55| 56                          LD      D,(HL)
        56| 210E00                      LD      HL,14
        59| 09                          ADD     HL,BC
        60| 73                          LD      (HL),E
        61| 2C                          INC     L
        62| 72                          LD      (HL),D
15:     63| CD4200              L4:     CALL    INTRP_REL+66
15:     66|                     ;exit code
        66| 9602                ;p-code RPU     2
```

The following listing is an example of function MAX translated on the 9900.

```
=============================================================
Final 9900 Code for Segment TEST  Procedure 2
Segment byte offset  30
                                        .RADIX  10
                              MP        .EQU    R9
                              SP        .EQU    R10
Source Object                 BK        .EQU    R12
P-Code n-Code                 SEG       .EQU    R13
(Dec. Offsets)                BASE      .EQU    R14
=============================================================

        0|                                .WORD   58,0
        0|
    4:  0| A8                  ;p-code   NATIVE
        1| A8                  ;p-code   NATIVE
        2| 8A69 000C 000A              C       @12(MP),@10(MP)
        8| 1105                        JLT     L1
       10| 1304                        JEQ     L1
    9: 12| CA69 000C 000E              MOV     @12(MP),@14(MP)
   11: 18| 1003                        JMP     L2
   13: 20| CA69 000A 000E    L1:       MOV     @10(MP),@14(MP)
   15: 26| 069C              L2:       BL      *BK
   15: 28|                   ;exit code
       28| 9602              ;p-code   RPU     2
```

The preceding listings show the hybrid mixture of P-code and n-code produced by the NCG. Cooperation between the n-code code and the P-machine interpreter is achieved using the following conventions:

* NATIVE is the P-code that instructs the interpreter to start executing n-code. On the Z80, 8080, and 8086, execution starts on the byte following the NATIVE instruction. On the 9900, execution begins on the first word boundary following the NATIVE instruction.

* The header lists the register conventions: P-machine registers on the left and processsor registers on the right.

* The following reference points on each processor listing indicate the instruction that returns the processor from n-code to P-code.

  Z80 Listing, line L4
  8086 Listing, line L3
  8080 Listing, line L4
  9900 Listing, line L2

* On the Z80 and 8080, global and external variables are referenced through BASE relative relocation. On the 8086, global variables are referenced through register DX, which contains Base. On the 9900, global variables are referenced indexed from R14, which contains Base. On both the 8086 and the 9900, external variables are referenced via base relative relocation.

On the whole, the listing looks very much like a listing created by the assembler. The following notes may help interpret the differences.

* P-code is preceded by the the notation, ;P-code (all other instructions are n-code.)

* The exit code point of the procedure is marked by the notation, ;exit code.

* The left-most column of numbers contains decimal byte offsets of equivalent P-code in the original code file. These offsets should help identify the source code by the offset in the compiler listing.

* The second column contains decimal byte offsets into the final procedure code generated by the NCG.

III.1.5 Limits of Native Code Generation

The NCG produces an object code file whose execution behavior is identical to the P-code file, except for differences in execution speed.

In those instances in which the compiler emits calls to a run-time support routine, the NCG leaves the P-code intact. Therefore, P-code is used in those places where translation would generate excessive code.

Sequences of straight n-code (code between a NATIVE instruction and its matching return instruction (See individual processor listings.) are treated by the p-machine as a single P-code. This fact causes two problems. First, although the <break> key may be recognized by the interpreter at any point, no further action is taken until the next P-code boundary (i.e.,until the current P-code is completed and the next P-code is encountered). Since there are no P-code boundaries in n-code, long sequences of n-code cannot be terminated by pressing the <break> key. Second, p-machine events (interrupts), like the break key, are only acted upon at P-code boundaries.

It is possible to work around these problems. The user may force a P-code procedure call by calling an empty procedure. Since the procedure call P-codes fall into the class of P-code that are never translated into n-code by the NCG, long sequences of n-code can be broken into smaller sequences by a procedure call. Since it is the procedure call itself that breaks up the sequence, the called procedure could be an empty shell.

Some unusual FORTRAN and Pascal contructs create code that the NCG will not translate. For example, using the Pascal primitive, P_Machine, to generate an RPU instruction and in FORTRAN, using a construct called an assigned GOTO construct will result in the error message: Undefined label.

III.2 Print Spooling

The print spooler is a program that allows the user to queue and print files concurrently with the normal execution of the p-System (while the console is waiting for input from the keyboard). The queue it creates is a file called *SYSTEM.SPOOLER, and the files the user wishes to print must reside on volumes that are on-line or an error will occur.

When SPOOLER is eX(ecuted, the following prompt line appears.

Spool: P(rint, D(elete, L(ist, S(uspend, R(esume, A(bort, C(lear, Q(uit

The following paragraphs define the prompt line options.

P(rint. Prompts for the name of a file to be printed. This name is then added to the queue. If SYSTEM.SPOOLER does not already exist, it is created. In the simplest case, P(rint may be used to send a single file to the printer. Up to 21 files may be placed in the print queue.

D(elete. Prompts for a file name to be taken out of the print queue. All occurrences of that file name are taken out of the queue.

L(ist. Displays the files currently in the queue.

S(uspend. Temporarily halts printing of the current file.

R(esume. Continues printing the current file after a S(uspend. R(esume also starts printing the next file in the queue after an error or an A(bort.

A(bort. Permanently stops the printing process of the current file and takes it out of the queue.

C(lear. Deletes all file names from the queue.

Q(uit. Exits the spooler utility and starts transferring files to the printer.

If an error occurs (e.g., a nonexistent file is specified in the queue), the error message appears only when the p-System is at the main system prompt line. If necessary, the spooler waits until the user returns to the outer level.

Program output to the printer may run concurrently with spooled output. The spooler finishes the current file and then turns the printer over to the user program. (The user program is suspended while it waits for the printer.) The user program should only do Pascal (or other high-level) writes to the printer. If the user program does printer output using unitwrite, the output is sent immediately and appears randomly interspersed with the spooler output.

The utility SPOOLER.CODE uses the operating system unit SPOOLOPS. Within this unit is a process called spooltask. Spooltask is started at boot time and runs concurrently with the rest of the UCSD p-System. The print spooler automatically restarts at boot time if *SYSTEM.SPOOLER is not empty. When the file *SYSTEM.SPOOLER exists, spooltask prints the files that it names. Spooltask runs as a background to the main operations of the p-System.

*SPOOLER.CODE interfaces with SPOOLOPS and uses routines within it to generate and alter the print queue within *SYSTEM.SPOOLER.

To restart the print spooling process if SPOOLER.CODE is executing when the system goes down, reboot the system, press X(ecute from the command prompt line, enter *SPOOLER.CODE, and press <return>. Then press R(esume.

III.3 CP/M XenoFile

This product is meant for users of CP/M 2.0 and later editions, or users who wish to access disks that contain a CP/M directory.

The p-System XenoFile package allows programs compiled on the p-System (in either Pascal, BASIC, or FORTRAN) to use disks that contain a CP/M directory To understand this chapter, read the CP/M Interface Guide, published by Digital Research, Inc.

XenoFile consists of three units and two utility programs. The utility program CPM_CNFIG must be run in order to configure the package for your particular system. CPM_FILER is a simple filehandler that allows you to read CP/M directories and transfer files on CP/M format disks to and from p-System format disks. Programs in UCSD Pascal may access CP/M disks by using CPM2_UNIT. FCPM provides the same facilities for FORTRAN-77 programs, and BCPM is for the use of BASIC programs.

III.3.1 Configuring the XenoFile Package

The first thing you must do is X(ecute CPM_CNFIG. This utility presents you with a series of questions about your disk configuration.

Your answers to these prompts are saved in a file called CPM_CNFIG.DATA. Every time CPM2_UNIT is initialized it reads and uses the information in CPM_CNFIG.DATA. Thus, every time you use XENOFILE, the file CPM_CNFIG.DATA must be on the prefixed disk.

CPM_FILER, FCPM, and BCPM all use CPM2_UNIT, so they also require CPM_CNFIG.DATA to be present on an on-line disk.

The following listing is the console display from a sample run of CPM_CNFIG (the values shown are the default values, which apply to many hardware configurations).

```
a) maximum drive number starting with 0                  = 3
b) highest physical track number                         = 76
c) number of sectors per track                           = 26
d) number of bytes per sector                            = 128
e) number of CP/M blocks per disk minus one              = 242
f) maximum random record number starting with 0          = 4095
g) buffer size minus one                                 = 127
h) maximum directory entries per disk                    = 63
i) number of blocks allocated for directory              = 2
j) offset to first CPM track                             = 2
k) number of records per block                           = 8
l) number of directory entries per sector                = 4
m) maximum record count for an extent                    = 128
n) maximum extent block index minus one                  = 15
o) sector interleave (CPM skew factor)                   = 6

CPM_CNFIG: Select letter ("a" to "o") of value to be changed,?
```

As the prompt shows, any of the values displayed may be changed in order to fit your particular hardware. When all of the values match your system, type "Q" to quit. When you type "Q", the bottom prompt line changes to:

```
QUIT: I(nstall changes, R(einstall defaults, E(xit without change
```

If you type "I", the values you chose are written to the file CPM_CNFIG.DATA (if this file did not exist before, CPM_CNFIG creates it). If you type "R", the default values are written to CPM_CNFIG.DATA. Finally, if you type "E", CPM_CNFIG terminates without producing any output, unless there is no CPM_CNFIG.DATA file on-line. In this case, E(xit creates a CPM_CNFIG.DATA file that contains the default values.

III.3.2 The CP/M Filer

The CPM_FILER program allows the user to perform four operations:

* Display a CP/M directory.
* Transfer a textfile from a CP/M disk to a p-System disk.
* Transfer a textfile from a p-System disk to a CP/M disk.
* Display a CP/M directory including all deleted entries.

When CPM_FILER is X(ecuted, the following prompt line appears:

CP/M Filer: D(ir list, C(P/M to Pascal, P(ascal to CP/M, ?

Typing "?" displays the remainder of the CP/M filer's prompts:

CP/M Filer: E(xtnd dir list, Q(uit, H(elp, ? , IV.0 [a.4]

There are six commands in the CP/M filer:

D List a CP/M directory.
C Transfer a CP/M textfile to a p-System textfile.
P Transfer a p-System textfile to a CP/M textfile.
E Display a CP/M directory, including deleted entries.
H Display some Help text.
Q Quit the CP/M filer and return to the main prompt line.

The transfer commands (C and P) use prompts that are comparable to prompts in the standard p-System Filer. They should be self-explanatory. If you have problems, refer to the help file by typing "H".

III.3.3 Using Xenofile CP/M Via UCSD Pascal

To use CP/M files directly from a program, the user must be familiar with the material covered in the CP/M Interface Guide by Digital Research.

The file CPM_CNFIG.DATA must be present on an on-line disk whenever a program uses CPM_FILER, CPM2_UNIT, FCPM, or BCPM. This file is created by CPM_CNFIG.CODE, as in preceding paragraphs.

To use the routines that access CP/M format disks, a UCSD Pascal program must contain the following declaration.

```
USES CPM2_UNIT;.
```

III.3.4 The File Control Block

The first thing we will describe is the Pascal version of the file control block (FCB). The information it contains is identical to a CP/M FCB, but it is declared in a manner that is easily used by Pascal programs. Here is the declaration of the FCB taken from the interface section of the unit CPM2_UNIT:

```
{ File control block and directory types }
fde_kind =
```

```
    (de_char, de_data, de_blks, de_cmpt);
  frec_num =
    0..CPM_rec_max;  {Random record number}
  fbyte_rng = 0..15;  {Dir entry byte index range}
  fword_rng = 0..7;  {Dir entry word index range}
  fblks =
    record
   { fil reserves space for byte fields: }
     fil: array [fword_rng] of integer;
   { xbm contains the extent's block map: }
     xbm: record case Boolean of
            true:
              (s: packed array [0..15] of byte);
            false:
              (l: array [0..7] of integer);
          end;
   end;
  fdirentry = record case fde_kind of
    de_char: (c: packed array [fbyte_rng] of char);
    de_data: (d: packed array [fbyte_rng] of byte);
    de_blks: (b: fblks);
    de_cmpt: (t: array [fword_rng] of Boolean);
    end;
  fcb = record
      de              { Copy of current dir entry for file }
                :fdirentry;
      cr              { Current file record (next rec to R/W) }
                      { Set by caller prior to R/W operation  }
                      { Set by CPM_ops sfirst and snext to DE }
                      {    number of last match, may be set by }
                      {    caller prior to a snext op, snext   }
                      {    advances cr before search starts    }
                :integer;
      rr              { Record to random read or write }
                :frec_num;
      ov              { Random record overflow, reset on Open }
                :Boolean;
      ro              { Read only file, (re)set during Open }
                :Boolean;
      dr              { CPM sys drive no, set by CPM_set_fname }
                :byte;
    end;
```

The FCB is divided into the following fields:

de Directory entry

cr Current record position in the current extent
rr Rdomor Record position
ov Overflow byte from random record
ro Read/Only Boolean
dr Drive number

de: fdirentry;

This is the directory entry field. It contains the same information as a CP/M directory, but the Pascal **type** fdirentry is organized as a variant record so that the programmer can deal with the various directory fields in a number of ways.

This is how the bytes are allocated in a CP/M directory:

Byte 0:	User number, must be in the range 0..16
Bytes 1..8:	Filename (ASCII characters)
Bytes 9..11:	File type (ASCII characters) In the file name and file type, each character also contains a flag bit.
Byte 12:	File extent. If this is greater than zero, this record is an extension of a previous FCB.
Bytes 13..14:	Unused; assumed to be zero
Byte 15:	Extent size in records (0..128)
Bytes 16..31:	Disk allocation map

These bytes may be accessed by a Pascal program in the following way. Assume that F is a variable of type FCB:

F.DE.C[n]	Allows access to the first 16 bytes as characters
F.DE.D[n]	Allows access to the first 16 bytes as bytes
F.DE.B[n]	Allows access to the second 16 bytes as either 16 bytes or 7 integers. These second 16 bytes are the blocks allocated to this file.
F.DE.B.S[n]	For the byte access
F.DE.B.L[n]	For the integer access
F.DE.T[n]	Allow access to the first 16 bytes as Booleans

cr: integer;

This integer is the current record position in this extent, and is used and maintained by the 'read' and 'write' operations. In addition, 'search first' and 'search next' use this location for holding the current directory index.

rr: integer;

This integer is used by 'read random', 'write random', 'compute file size', and 'set random record' to maintain the current position within the file.

ov: Boolean;

This Boolean is used to indicate an overflow in a random file access.

dr: byte;

This byte is the number of the drive that contains the file. It is set by CPM_SET_FNAME. This field is not explicitly named in CP/M: it corresponds to the first byte of the de field in a CP/M FCB.

III.3.5 File Control Block Intrinsics

While a Pascal program may use the FCB structures directly, CPM2_UNIT also contains five routines for handling the FCB, which are described in this section:

```
Function CPM_SET_FNAME
Function CPM_GET_FNAME
Function CPM_GET_FLAG
Procedure CPM_SET_FLAG
Procedure CPM_CLR_FLAG
```

First, a note about error conditions for the functions described in this section (and also in the following section). Each of the functions in CPM2_UNIT returns an ioresult value. If this is zero, the function was successful. If this is not zero, some error occurred. The standard error numbers are shown in the Users Manual, Version IV.0. Three new ioresults have also been introduced:

32 Attempt to write to a read-only file
33 A random read from a nonexistent record
34 The current record is out of range

Since the CPM2_UNIT functions return the ioresult value, the program does not need to turn off I/O checking in order to determine whether a call was successful.

Function CPM_SET_FNAME
(**var** F: FCB; NAME: string): integer;

Stores NAME into the FCB and initializes the file for opening.

This function handles all valid CP/M wild card conventions and drive specifications. If the file name is not valid, the function returns a result of 7.

Examples:

```
  { use default drive: }
RSLT := CPM_SET_FNAME(F,DDT.COM);
  { use Drive A: }
RSLT := CPM_SET_FNAME(F,A:SID.COM);
  { both types of wild cards: }
RSLT := CPM_SET_FNAME(F,?:W?.*);
```

Function CPM_GET_FNAME
(**var** F: FCB; **var** NAME: string): integer;

This function returns the name and extent fields of the FCB as a string. The result of the function is ALWAYS 0.

Examples:

```
RSLT := CPM_SET_FNAME(F,'DDT.COM');
RSLT := CPM_GET_FNAME(F,NAME);
  { NAME = 'DDT      .COM' }

RSLT := CPM_SET_FNAME(F,'A:SID.COM');
RSLT := CPM_GET_FNAME(F,NAME);
  { NAME = 'SID      .COM' }

RSLT := CPM_SET_FNAME(F,'?:W?.*');
RSLT := CPM_GET_FNAME(F,NAME);
  { NAME = 'W?       .???' }
```

Function CPM_GET_FLAG
(**var** F: FCB; FLAG: integer): Boolean;

For each of the 11 bytes that make up the file name and file extent, bit 7 is an attribute flag. This function returns the value of that flag. The FLAG parameter must be in the range 1..11, specifying one of the 11 bytes.

These are the current CP/M interpretations of the various flags:

1..4	undefined
5..8	undefined but reserved
9	file is read only
10	file is a System file (not displayed by the DIR command)
11	undefined but reserved

Procedure CPM_SET_FLAG
(**var** F: FCB; FLAG:integer);

This procedure allows the user to set one of the flags, and is used in conjunction with the ATTRIBUTE function (which is described in the following section).

Example:

```
CPM_SET_FLAG(F,9); { set the read only bit }
RSLT := CPM(F,CPM_ATTRIB);
```

Procedure CPM_CLR_FLAG
(**var** F: FCB; FLAG: integer);

This procedure allows the user to clear one of the flags, and is used in conjunction with the ATTRIBUTE function (which is described in the following section).

Example:

```
CPM_CLR_FLAG(F,9); (* SET READ/ONLY BIT *)
RSLT := CPM(F,CPM_ATTRIB);
```

III.3.6 CP/M Interface Entry Points

In addition to the routines described above, CPM2_UNIT provides a number of routines that can be called in ways that correspond to the entry points of the CP/M Interface. This section describes the way to call each entry point, in the order given in the CP/M Interface Guide.

For each CP/M Interface entry point, we show the corresponding Pascal routine call, and two examples: a typical call from CP/M assembly language, and a typical call from a Pascal program.

Some of the Pascal calls to CP/M Interface entry points are individual procedures, but many of them are calls to the one function CPM. A call to CPM specifies an FCB, and a particular operation, for example:

```
result:= CPM(F, CPM_OPEN);  {opens a file}
result:= CPM(F, CPM_CLOSE);  {closes a file}
```

The program examples use these conventions:

F is an FCB declared as:

```
F: FCB;    {in Pascal}      {... or ...}
F:         DS        36    ;in assembly language
```

RSLT is an integer defined as follows:

```
RSLT: integer;      { in Pascal }
RSLT:   DB          0     ;in assembly language
```

Ambiguous is a generic name for file names and file types that can be specified by the CP/M wild card characters. A question mark in certain FCB fields indicates that any value is valid at that location, and an asterisk in a file name or file type specification causes the field to be filled with question marks. For example:

```
CPM_SET_FNAME(F,'A:AFILE?.COM');
    {... name and attribute set to: 'AFILE?   COM'}
CPM_SET_FNAME(F,'*.*');
    {... name and attribute set to: '???????????'}
CPM_SET_FNAME(F,'FILE*.*');
    {... name and attribute set to: 'FILE???????'}
CPM_SET_FANME(F,'?:THEFILE.COM');
    { user to '?' for search first or next }
```

As we explained above, every **function** in CPM2_UNIT returns an ioresult value. This equals zero if no error occurred. The meanings of ioresult values are listed in the Users Manual, Version IV.0. There are also three new codes:

32 attempt to write to a read only file
33 attempt to read a nonexistent record
34 current record is out of range

Finally, CP/M uses the term "vector" to refer to an array of bits. In CPM2_UNIT, all vectors are declared as sets. These include DRV_VECTOR, RO_VECTOR, and the ALLOC_VECTOR. We will discuss them where necessary.

FUNCTION 12: Return the version number

Pascal definition:

```
Function CPM_VER_NUMBER: integer;
```

Returns the version number.

This returns the current version of CPM2_UNIT. To be compatible with CP/M, it returns a value of greater than 020h (32 decimal) but less than 100h (256). Because this is compatible with version 2.x, it means that CPM2_UNIT supports random access. Note that currently all of the CP/M 2.x functions are implemented except function 31: Get disk parameters.

CP/M Example:

```
;RETURN THE CPM VERSION
;NUMBER
MVI     C,12
CALL    BDOS
STA     VER$NUMBER
```

Pascal Example:

```
VER_NUMBER := CPM_VER_NUMBER;
```

FUNCTION 13: Reset the disk system

Pascal definition:

```
Procedure CPM_DSK_RESET;
```

This procedure resets CPM2_UNIT's values to a known state:

1) All disks are set to read/write
2) Drive A: becomes the default drive
3) The default buffer becomes the current buffer

CP/M example:

```
;RESET THE BDOS
MVI     C,13
CALL    BDOS
```

Pascal example:

```
CPM_DSK_RESET;
```

FUNCTION 14: Select disk

Pascal definition:

```
Function CPM_SELECT (DRIVE: integer): integer;
```

Returns the error status.

This function selects DRIVE as the new default drive. All subsequent disk operations where the dr field of the FCB is zero refer to this drive. Note that this function DOES return an error status, though CP/M itself does not.

CP/M example:

```
;SELECT A NEW DEFAULT DRIVE
MVI     E,DRIVE             ;E = NEW DRIVE
CALL    C,14
CALL    BDOS                ;NO ERRORS
```

Pascal example:

```
RSLT := CPM_SELECT(DRIVE);
IF RSLT <> 0 THEN
   (* error *)
```

FUNCTION 15: Open file(s)

Pascal definition:

```
CPM (F, CPM_OPEN)
```

Returns the error status.

This function activates an existing file in the directory of the currently active user. The file name and file type may be ambiguous ('?'). Normally there are no question marks.

NOTE: The directory code is not returned as in CP/M.

CP/M example:

```
        LXI     D,F
        MVI     C,15
        CALL    BDOS
        CPI     0FFH
        JNZ     OK                  ;JUMP IF NO ERRORS

F:      DB      0,"AFILE   COM",0,0,0,0
        DS      20
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'AFILE.COM');
IF RSLT <> 0 THEN
  (* BAD FILE NAME *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_OPEN);
    IF RSLT <> 0 THEN
      (* FILE DOES NOT EXIST *)
  END;
```

FUNCTION 16: Close file

Pascal definition:

```
CPM (F, CPM_CLOSE)
```

Returns the error status.

This function closes a file that was previously opened by a CPM_OPEN or CPM_MAKE operation. The file name and type are matched as with CPM_OPEN (See preceding Pascal example).

Files opened only for reading do not need to be closed, but a file that was also written must be closed in order to update the directory.

NOTE: The directory index is not returned, as it would be under CP/M.

CP/M example:

```
        LXI     D,F
        MVI     C,16
        CALL    BDOS
        CPI     0FFH
        JNZ     OK                  ;JUMP IF NO ERRORS
F:      DB      0,"AFILE   COM",0,0,0,0
```

Pascal example:

```
RSLT := CPM(F,CPM_CLOSE);
IF RSLT <> 0 THEN
  (* COULD NOT CLOSE *)
```

FUNCTION 17: Search for first

Pascal definition:

```
CPM (F, CPM_SFIRST)
```

Returns the error status.

This function searches the directory for a file that matches the FCB. If a match is found, the directory entry is returned in CPM_F, a global FCB used by search first and search next for the purpose of returning directory entries.

NOTE: This is very different from the way CP/M operates, but the net result is the same.

The FCB may contain ambiguous references as in open, but in addition byte zero of the FCB may be a question mark ('?' = 63). If byte zero is a question mark, then the auto select function is disabled, the default disk is searched and every directory entry is checked, regardless of whether it is deleted or not, or which user number it belongs to.

NOTE: Byte zero may be set to a question mark by using CPM_SET_FNAME with the drive specification a question mark, as in the example below. This allows a program to interrogate every directory entry when used in conjunction with function 18, search next.

CP/M example: look at each of the entries in the directory.

```
        ;SET DMA ADDRESS
        LXI     D,DIRBUF
        MVI     C,26
        CALL    BDOS                ;SET DMA ADDRESS
                                    ; TO DIRBUF

        ;LOOK FOR FIRST ENTRY
        LXI     D,F
        MVI     C,17
        CALL    BDOS
        CPI     0FFH
        JZ      NOMATCH             ;JUMP IF NO ENTRY

        ;LOOK AT ENTRY WHICH IS IN DIRBUF[A*32] ....
        LOOK:

        ;SEARCH THE NEXT ENTRY
        LXI     D,F
        MVI     C,18
        CALL    BDOS
        CPI     0FFH
        JNZ     LOOK                ;JUMP IF FOUND

        NOMATCH:
        ;EXIT

F:      DB      '?','???????????','?','?','?',0
        DS      20
        DIRBUF: DS                  128
```

Pascal example:

```
RSLT := CPM_SET_FNAME('?:*.*');
                    (* USER AND NAME AND TYPE = '?' *)
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    FOR I := 12 TO 14 DO
      FCB.DE.C[I] := '?'; (* SET EX,S1,S2 TO '?' *)
    RSLT := CPM(F,CPM_SFIRST);
    WHILE RSLT = 0 DO
      BEGIN
        (* LOOK AT DIRECTORY IN CPM_F *)
        FOR I := 1 TO 8 DO
          WRITE(CPM_F.DE.C[I]);
        WRITE('.');
        FOR I := 9 TO 11 DO
          WRITELN(CPM_F.DE.C[I]);

        (* SEARCH FOR THE NEXT MATCH *)
        RSLT := CPM(F,CPM_SNEXT);
      END;
  END;
```

FUNCTION 18: Search for next

Pascal definition:

```
CPM (F, CPM_SNEXT)
```

Returns the error status.

This function searches the directory for a file that matches the FCB, starting from the directory entry in F.CR + 1. Normally, the cr field is set by search first as in the preceding example. search next is simply a continuation of search first, and operates in the same manner. If a match is found, the directory entry is returned in CPM_F as in search first.

Please refer to search first (function 17) for an example.

FUNCTION 19: Delete file

Pascal definition:

```
CPM (F, CPM_DELETE)
```

Returns the error status.

This function deletes a file entry from the directory. The name or type may be ambiguous.

CP/M example:

```
        LXI     D,F                 ;DE = FCB ADDRESS
        MVI     C,19
        CALL    BDOS                ;DELETE THE FILE
        CPI     0FFH
        JZ      NOT$DELETED         ;JUMP IF UNSUCCESSFUL
F:      DB      0,"AFILE   COM",0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'AFILE.COM');
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_DELETE);
    IF RSLT <> 0 THEN
      (* ERROR *)
  END;
```

FUNCTION 20: Read a file sequentially

Pascal definition:

```
CPM (F, CPM_RDSEQ)
```

Returns the error status.

This function reads one record from the file, at the position defined by F.CR, into the buffer pointed to by CPM_BUFF. The CR field is then incremented. If the CR field overflows, the next extent in the directory is automatically opened and the CR field is set to 0.

NOTE: If an attempt is made to read past the end of the file, the function returns an ioresult of 33 (record does not exist) to signify end of file.

CP/M example: Read the contents of a file.

```
          ;OPEN THE FILE
          LXI     D,F
          MVI     C,15
          CALL    BDOS
          CPI     0FFH
          JZ      ERROR             ;JUMP IF UNABLE TO OPEN

          ;SET THE DMA ADDRESS
          LXI     D,BUFFER
          MVI     C,26
          CALL    BDOS

          ;READ THE FILE
          RDLOOP:
          LXI     D,F
          MVI     C,20
          CALL    BDOS              ;READ NEXT RECORD
          CPI     0
          JNZ     ERROR             ;JUMP IF END OF FILE

          ;DO SOMETHING

          JMP     RDLOOP
F:        DB      0,"AFILE      ",0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'AFILE');
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_OPEN);
    IF RSLT <> 0 THEN
      (* ERROR *)
    ELSE
      BEGIN
        NEW(CPM_BUFF); (* ALLOCATE A BUFFER *)
        REPEAT
          RSLT := CPM(F,CPM_RDSEQ);
          IF RSLT <> 0 THEN
```

```
            (* ERROR *)
          ELSE
            (* DO SOMETHING *)
        UNTIL RSLT <> 0;
      END;
END;
```

FUNCTION 21: Write a file sequentially

Pascal definition:

```
CPM (F, CPM_WRSEQ)
```

Returns the error status.

This function writes one record to the file, at the position defined by F.CR, from the buffer pointed to by CPM_BUFF. The CR field is then incremented. If the CR field overflows, the next extent in the directory is automatically opened, or made, and the CR field is set to 0.

CP/M example: Write a record to a new file

```
;OPEN THE FILE
LXI     D,F
MVI     C,22            ;CPM_MAKE
CALL    BDOS
CPI     0FFH
JZ      ERROR           ;JUMP IF UNABLE TO MAKE

;SET THE DMA ADDRESS
LXI     D,BUFFER
MVI     C,26
CALL    BDOS

;WRITE TO THE FILE
LXI     D,F
MVI     C,21
CALL    BDOS            ;WRITE THE RECORD
CPI     0
JNZ     ERROR           ;JUMP IF ERROR

;CLOSE THE FILE
LXI     D,F
MVI     C,16
CALL    BDOS
```

```
        CPI     0FFH
        JZ      ERROR           ;JUMP IF ERROR
F:      DB      0,'NEWFILE     ',0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'NEWFILE');
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_MAKE);
    IF RSLT <> 0 THEN
      (* ERROR *)
    ELSE
      BEGIN
        NEW(CPM_BUFF); (* ALLOCATE A BUFFER *)
        RSLT := CPM(F,CPM_WRSEQ);
        IF RSLT <> 0 THEN
          (* ERROR *)
        RSLT := CPM(F,CPM_CLOSE);
        IF RSLT <> 0 THEN
          (* ERROR *)
      END;
  END;
```

FUNCTION 22: Make a new file

Pascal definition:

```
CPM (F, CPM_MAKE)
```

Returns the error status.

This function creates a new file entry in the directory, and opens the file. The file name must be unambiguous, and cannot already be in use. The program may ensure this by performing an open operation before the make.

CP/M example:

```
;CHECK OPEN TO CHECK FOR DUPLICATE FILE
        LXI     D,F
        MVI     C,15
        CALL    BDOS            ;OPEN
        CPI     0FFH
```

```
        JNZ     MAKEIT          ;JUMP IF DOES NOT EXIST

        ;DELETE
        LXI     D,F
        MVI     C,19
        CALL    BDOS            ;DELETE
        CPI     0FFH
        JZ      ERROR           ;SHOULD NOT HAPPEN
                                ; UNLESS MEDIA ERROR

        ;MAKE THE FILE
        MAKEIT:
        LXI     D,F
        MVI     C,22
        CALL    BDOS            ;MAKE
        CPI     0FFH
        JZ      ERROR
F:      DB      0,'NEWFILE     ',0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'NEWFILE');
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_OPEN);
    IF RSLT <> 0 THEN
      (* ERROR *)
    ELSE
      BEGIN
        RSLT := CPM(F,CPM_DELETE);
        IF RSLT <> 0 THEN
          EXIT(); (* ERROR bomb off *)
      END;
    RSLT := CPM(F,CPM_MAKE);
    IF RSLT <> 0 THEN
      (* ERROR *)
  END;
```

FUNCTION 23: Rename file

Pascal definition:

```
Function CPM_RENAME (F, S: string):integer;
```

Returns the error status.

This function renames a file in the directory. The names are assumed to be unambiguous. To rename a file, first use CPM_SET_FNAME to set up F with the old name, then call CPM_RENAME with the new name.

CP/M example:

```
        LXI     D,F
        MVI     C,23
        CALL    BDOS            ;RENAME
        CPI     0FFH
        JZ      ERROR           ;JUMP IF ERROR

F:      .DB     0,"OLDFILE     ",0,0,0,0
        .DB     0,"NEWFILE     ",0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'OLDFILE');
IF RSLT <> 0 THEN
   (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM_RENAME(F,'NEWFILE');
    IF RSLT <> 0 THEN
      (* ERROR *)
  END
```

FUNCTION 24: Return login vector

Pascal definition:

```
Procedure CPM_GET_DRIVES
    (var DRV_VECTOR: BIT_VEC);
```

This procedure returns in DRV_VECTOR a set of 16 elements numbered 0..15. Each element corresponds to one of the 16 possible drives and is **in** the set if that drive is currently logged in.

CP/M example: Is drive 0 logged in?

```
        MVI     C,24
        CALL    BDOS
        MOV     A,L
        RRC                     ;MOVE BIT 0 TO CARRY
        JCC     NOT$LOGGED      ;JUMP IF NOT LOGGED IN
```

Pascal example:

```
CPM_GET_DRIVES(DRV_VECTOR);
IF 0 IN DRV_VECTOR THEN
  (* DO SOMETHING *)
```

FUNCTION 25: Return current disk

Pascal definition:

```
Function CPM_GET_DSK: integer;
```

Returns the current disk number.

This function returns the number of the currently selected disk. When an operation is done and F.DR equals zero, then the currently selected disk is the one that the operation affects.

The number returned is in the range 0..15, corresponding to the drive names A: .. P:.

CP/M example:

```
        MVI     C,25
        CALL    BDOS
```

```
        STA     CUR$DSK
```

Pascal example:

```
CUR_DSK := CPM_GET_DSK;
```

FUNCTION 26: Set DMA address

Pascal definition:

```
CPM_BUFF := CPM_BUFFP;
```

This variable defined in the INTERFACE section of CPM2_UNIT is a pointer to a buffer where the next file operation will take place. At initialization time, CPM_BUFF points to a default buffer new'ed onto the heap. Also, CPM_BUFF is set back to CPM_BUFF_DFLT during CPM_RESET.

NOTE: CPM_BUFF_DFLT is also a variable defined in the **interface** section of CPM2_UNIT, and the user may set CPM_BUFF equal to it at any time.

CP/M example:

```
        LXI     D,NEWBUFFER
        MVI     C,26
        CALL    BDOS
```

Pascal example:

```
NEW(NEWBUFFER);
CPM_BUFF := NEWBUFFER;
```

FUNCTION 27: Get allocation vector

Pascal definition:

```
procedure CPM_GET_ALLOC
   (var ALLOC_VECTOR: BLK_VEC);
```

This procedure returns a set that contains all currently available blocks on the disk.

NOTE: If the currently selected disk is write protected, then the allocation map is invalid. This procedure is not normally used by applications programs.

CP/M example:

```
MVI     C,27
CALL    BDOS
SHLD    ALLOC$ADR       ;STORE ADDRESS OF VECTOR
```

Pascal example:

```
CPM_GET_ALLOC(ALLOC_VEC);
```

FUNCTION 28: Write protect disk

Pascal definition:

```
procedure CPM_WP (DRIVE: integer);
```

This procedure write protects the specified drive. The number passed must be in the range 0..15. All subsequent attempts to write to that disk cause the function to return an ioresult of 16.

CP/M example: write protect drive B:

```
;SELECT DRIVE B
MVI     E,1
MVI     C,14
CALL    BDOS            ;SELECT DRIVE B
MVI     C,28
CALL    BDOS            ;WRITE PROTECT
```

Pascal example:

```
CPM_WP(1);
```

FUNCTION 29: Get read/only vector

Pascal definition:

```
procedure CPM_GET_RO (RO_DRIVES: BIT_VEC);
```

This procedure returns a set with 16 (0..15) elements: these indicate which disk drives are currently marked as read-only. A file may be write protected explicitly by a call to CPM_WP, or CPM2_UNIT may automatically declare a drive as write protected if the medium was changed without the drive being logged in.

CP/M example: Is drive A: read only?

```
        MVI     C,29
        CALL    BDOS
        MOV     A,L
        RRC                     ;BIT 0 TO CARRY
        JNC     NOT$RO          ;JUMP IF BIT = 0
```

Pascal example:

```
CPM_GET_RO(DRV_VECTOR);
IF 0 IN DRV_VECTOR THEN
  (* IS READ ONLY *)
```

FUNCTION 30: Set file attributes

Pascal definition:

```
CPM (F, CPM_ATTRIB)
```

Returns the error result.

This function is used with the new procedures CPM_SET_FLAG and CPM_CLR_FLAG described above. They allow the programmer to alter the bit flags maintained in bit 7 of the 11 file name and file type bytes in a directory entry. The file name and file type must not be ambiguous.

CP/M example: Set a file to read-only

```
        LXI     D,F
        LXI     H,9
        DAD     D               ;HL POINTS AT THE
                                ; TYPE BYTE
        MOV     A,M
```

```
        ORI     080H            ;SET BIT 7 OF TYPE BYTE 1
        MOV     M,A
        MVI     C,30
        CALL    BDOS            ;CALL BDOS TO SET
                                ; THE ATTRIBUTE

F:      DB      0,'AFILE      ',0,0,0,0
        DB      20
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'AFILE');
IF RSLT <> 0 THEN
   (* ERROR *)
ELSE
  BEGIN
    CPM_SET_FLAG(F,9);
    RSLT := CPM(F,CPM_ATTRIB);
    IF RSLT <> 0 THEN
      (* ERROR *)
  END;
```

FUNCTION 31: Get disk parameters

NOT IMPLEMENTED

FUNCTION 32: Set/get user code

Pascal definition:

```
CPM_USER: BYTE;
```

This is a variable defined in the **interface** section of CPM2_UNIT. This variable defines which user is currently valid when searching the directory for matches to an FCB.

CP/M example: Set the user code to 15

```
        MVI     E,15
        MVI     C,32
        CALL    BDOS
```

Pascal example:

```
CPM_USER := 15;
```

CP/M example: Get the current user code

```
        MVI     E,0FFH
        MVI     C,32
        CALL    BDOS
        STA     CURR$USER
```

Pascal example:

```
CURR_USER := CPM_USER;
```

FUNCTION 33: Read random

Pascal definition:

```
CPM (F, CPM_RDRND)
```

Returns the error result.

This function reads a random record from a file. The RR field in the FCB must have been set to the desired record number. If the data is in the file, the result returned is zero, and the data are stored at the current DMA address pointed to by CPM_BUFF. If there is currently no data at that position, the function returns a 33.

If upon returning from the function the OV field of the FCB is true, then the RR field was too large and the function returns a result of 8.

NOTE: Sequential reads may be commenced from this point, but since random reading does not automatically increment the RC field in the FCB, the current record is reread. The user may simulate sequential reading by simply incrementing the RR field and calling this function.

CP/M example: Read record 1000.

```
        ;ASSUME THE FILE IS ALREADY OPEN
        ;SET R0,R1,R2 TO 1000
        LXI     H,F
        LXI     D,33
        DAD     D                       ;HL POINTS TO R0
        LXI     D,1000
```

```
MOV     M,E             ;STORE R0
INX     H
MOV     M,D             ;STORE R1
INX     H
MVI     M,0             ;ZERO R2

;SET DMA ADDRESS
LXI     D,BUFFER
MVI     C,26
CALL    BDOS

;READ THE RECORD
LXI     D,F
MVI     C,33
CALL    BDOS
CPI     0
JNZ     ERROR           ;JUMP IF ERROR
```

Pascal example:

```
F.RR := 1000;
CPM_BUFF := BUFFER;
RSLT := CPM(F,CPM_RDRND);
IF RSLT <> 0 THEN
    (* ERROR *)
```

FUNCTION 34: Write random

Pascal definition:

```
CPM (F, CPM_WRRND)
```

Returns the error result.

This function writes a random record to a file. The RR field in the FCB must be set to the desired record number. The file is automatically extended as neccessary to accommodate this record, as long as there is room on the disk and in the directory.

If upon returning from the function, the OV field of the FCB is true, then the RR field was too large, and the function restums a result of 8.

NOTE: As in random reading, sequential writing is allowed following a random write, but since the RR field is not incremented after a random write, the first sequential write will be to the current position. Sequential writing may be simulated by explicitly incrementing the RR field.

CP/M example: Write record 1000.

```
;ASSUME THE FILE IS ALREADY OPEN
;SET R0,R1,R2 TO 1000
LXI     H,F
LXI     D,33
DAD     D               ;HL POINTS TO R0
LXI     D,1000
MOV     M,E             ;STORE R0
INX     H
MOV     M,D             ;STORE R1
INX     H
MVI     M,0             ;ZERO R2

;SET DMA ADDRESS
LXI     D,BUFFER
MVI     C,26
CALL    BDOS

;WRITE THE RECORD
LXI     D,F
MVI     C,34
CALL    BDOS
CPI     0
JNZ     ERROR           ;JUMP IF ERROR
```

Pascal example:

```
F.RR := 1000;
CPM_BUFF := BUFFER;
RSLT := CPM(F,CPM_WRRND);
IF RSLT <> 0 THEN
    (* ERROR *)
```

FUNCTION 35: Compute file size

Pascal defintion:

```
CPM (F, CPM_FSIZE)
```

Returns the error result.

This function computes the size of a file and returns the size in the RR field of the FCB. The file name and type must not be ambiguous. If the file is too large then the OV field will be set to true, and the function returns an 8.

CP/M example:

```
        LXI     D,F
        MVI     C,35
        CALL    BDOS            ;NO ERRORS IN CP/M THE
                                ; BDOS BOMBS
F:      DB      0,"AFILE      ",0,0,0,0
```

Pascal example:

```
RSLT := CPM_SET_FNAME(F,'AFILE');
IF RSLT <> 0 THEN
  (* ERROR *)
ELSE
  BEGIN
    RSLT := CPM(F,CPM_FSIZE);
    IF RSLT <> 0 THEN
      (* ERROR *)
  END;
```

FUNCTION 36: Set random record

Pascal defintion:

```
CPM (F, CPM_SETRR)
```

Returns the error result.

This function sets the RR field of the FCB to the current random record position of a file which is already open. This allows the caller to save the current position in the file for later access by a random read or write. If the file is too large, then the OV field will be true, and the function returns an 8.

CP/M example:

```
        ;ASSUME THE FILE IS OPEN
        LXI     D,F
        MVI     C,36
        CALL    BDOS                ;NO ERRORS
```

Pascal example:

```
(* ASSUME THE FILE IS OPEN *)
RSLT := CPM(F,CPM_SETRR);
IF RSLT <> 0 THEN
  (* ERROR *)
```

III.3.7 Summary: CP/M Interface -- Calls to CPM2_UNIT

The following table is a quick summary of the correspondence between CP/M BDOS calls 12 through 36, and CPM2_UNIT calls:

Call #	CP/M Call Sequence		p-System Sequence
	;RETURN THE CPM VERSION		
	;NUMBER		
12	MVI	C,12	VER_NUMBER := CPM_VER_NUMBER;
	CALL	BDOS	
	STA	VER$NUMBER	
	;RESET THE BDOS		
13	MVI	C,13	CPM_DSK_RESET;
	CALL	BDOS	
	;SELECT A NEW DEFAULT		
	;DRIVE		
14	MVI	E,DRIVE	RSLT := CPM_SELECT(DRIVE:INTEGER);
	CALL	C,14	
	CALL	BDOS	
	;FILE OPEN		
15	LXI	D,F	RSLT := CPM(F,CPM_OPEN);
	MVI	C,15	
	CALL	BDOS	
	;FILE CLOSE		
16	LXI	D,F	RSLT := CPM(F,CPM_CLOSE);
	MVI	C,16	
	CALL	BDOS	
	;SEARCH FOR FIRST		
	;OCCURRENCE OF FILE(s)		
17	LXI	D,F	RSLT := CPM(F,CPM_FIRST);
	MVI	C,17	
	CALL	BDOS	

```
        ;SEARCH FOR NEXT
        ;OCCURRENCE OF FILE(s)
18      LXI     D,F             RSLT := CPM(F,CPM_NEXT);
        MVI     C,18
        CALL    BDOS

        ;FILE DELETE
19      LXI     D,F             RSLT := CPM(F,CPM_DELETE);
        MVI     C,19
        CALL    BDOS

        ;READ SEQUENTIAL
20      LXI     D,F             RSLT := CPM(F,CPM_RDSEQ);
        MVI     C,20
        CALL    BDOS

        ;WRITE SEQUENTIAL
21      LXI     D,F             RSLT := CPM(F,CPM_WRSEQ);
        MVI     C,21
        CALL    BDOS

        ;FILE MAKE
22      LXI     D,F             RSLT := CPM(F,CPM_MAKE);
        MVI     C,22
        CALL    BDOS

        ;FILE RENAME
23      LXI     D,F             RSLT := CPM(F,CPM_RENAME);
        MVI     C,23
        CALL    BDOS

        ;RETURN LOGIN VECTOR
24      MVI     C,24            CPM_GET_DRIVES(LOGIN);
        CALL    BDOS
        SHLD    LOGIN
```

```
        ;RETURN CURRENT DISK
25      MVI     C,25            CURR_DISK := CPM_GET_DSK;
        CALL    BDOS
        STA      CURR$DISK

        ;SET ADDRESS OF NEXT I/O
        ;OPERATION
26      LXI     D,BUFF          NEW(CPM_BUFF);
        MVI     C,26                or
        CALL    BDOS            CPM_BUFF := BUFF_ADDRESS;

        ;GET THE ALLOCATION MAP FOR
        ;THE CURRENT DEFAULT DISK
27      MVI     C,27            CPM_GET_ALLOC(ALLOC);
        CALL    BDOS
        SHLD    ALLOC$ADR

        ;WRITE PROTECT CURRENT
        ;DISK
28      MVI     E,DISK          CPM_WP(DISK);
        MVI     C,14
        CALL    BDOS
        MVI     C,28
        CALL    BDOS

        ;GET READ/ONLY VECTOR
29      MVI     C,29            CPM_GET_RO(RO_DRIVES);
        CALL    BDOS
        SHLD    RO$DRIVES

        ;SET FILE ATTRIBUTES
30       MVI    C,30            RSLT := CPM(F,CPM_ATTRIB);
        CALL    BDOS
```

```
        ;GET DISK PARMS
31      MVI     C,31            NOT SUPPORTED
        CALL    BDOS

        ;SET/GET USER CODE
        ;SET
32      MVI     E,15            CPM_USER := 15
        MVI     C,32
        CALL    BDOS

        ; GET
        MVI     E,0FFH          USER := CPM_USER
        MVI     C,32
        CALL    BDOS

        ;READ RANDOM
33      LXI     D,F             RSLT := CPM(F,CPM_RDRND);
        MVI     C,33
        CALL    BDOS

        ;WRITE RANDOM
34      LXI     D,F             RSLT := CPM(F,CPM_WRRND);
        MVI     C,34
        CALL    BDOS

        ;COMPUTE FILE SIZE
35      LXI     D,F             RSLT := CPM(F,CPM_FSIZE);
        MVI     C,35
        CALL    BDOS

        ;SET RANDOM RECORD
36      LXI     D,F             RSLT := CPM(F,CPM_SETRR);
        MVI     C,36
        CALL    BDOS
```

III.3.8 FORTRAN Interface to CP/M XenoFile

This section explains how p-System FORTRAN programs can access CP/M disks. The user must be familiar with the information in this supplement and the CP/M Interface Guide.

A FORTRAN program invokes XenoFile by using the unit FCPM. This unit should be installed in *SYSTEM.LIBRARY so that it can be found during compilation and execution. The FORTRAN program must then include the directive "$USES FCPM" in order to use the routines provided by FCPM. Since all of the routines, including all of the integer functions in FCPM, start with the letter "C", it is also desirable to include the statement "IMPLICIT INTEGER (C)" in the FORTRAN program.

A correspondence between CP/M BDOS calls 12 through 36 and FCPM calls follows:

Call #	FORTRAN p-System sequence	Description
12	RSLT = CPM12()	Return the version number
13	CALL CPM13	Reset the disk system
14	RSLT = CPM14(DRIVE)	Select disk
15	RSLT = CPMI(F,15)	Open file
16	RSLT = CPMI(F,16)	Close file
17	RSLT = CPMI(F,17)	Search for first
18	RSLT = CPMI(F,18)	Search for next
19	RSLT = CPMI(F,19)	Delete file
20	RSLT = CPMI(F,20)	Read a file sequentially
21	RSLT = CPMI(F,21)	Write a file sequentially
22	RSLT = CPMI(F,22)	Make a new file
23	RSLT = CPM23(F,NAME)	Rename file
24	CALL CPM24(LOGVEC)	Return login vector
25	RSLT = CPM25()	Return current disk
26	CALL CPM26G(B)	Get CPM buffer
	CALL CPM26P(B)	Put CPM buffer
27	CALL CPM27(ALLVEC)	Get allocation vector
28	CALL CPM28(DRIVE)	Write protect disk
29	CALL CPM29(RODVEC)	Get read/only vector
30	RSLT = CPMI(F,30)	Set file attributes
31	NOT SUPPORTED	
32	RSLT = CPM32G()	Get user code
	CALL CPM32S(USRNUM)	Set user code
33	RSLT = CPMI(F,33)	Read random
34	RSLT = CPMI(F,34)	Write random
35	RSLT = CPMI(F,35)	Compute file size
36	RSLT = CPMI(F,36)	Set random record

The following additional FORTRAN calls with no corresponding CP/M BDOS function calls are also available through FCPM:

CALL CPMSFN(F,NAME)	Set file name
CALL CPMGFN(F,NAME)	Get file name
RSLT = CPMGFL(F,FLAG)	Get attribute flag
CALL CPMSFL(F,FLAG)	Set attribute flag to 1
CALL CPMCFL(F,FLAG)	Clear attribute flag to 0
CALL CPMFCB(G)	Get FCB for calls 17 & 18
RSLT = CPMGE(F)	Get file extent number
CALL CPMSE(F,EXTENT)	Set file extent number
RSLT = CPMGCR(F)	Get current record number
CALL CPMSCR(F,CRNUM)	Set current record number
RSLT = CPMGRR(F)	Get random record number
CALL CPMSRR(F,RRNUM)	Set random record number
RSLT = CPMGB(B,BYTNUM)	Get byte out of buffer (Used after calling CPM26G)
CALL CPMSB(B,BYTNUM,VALUE)	Set byte in buffer (Used before calling CPM26P)

The following items define the arguments for the preceding calls.

RSLT. An integer value returned by the function.

DRIVE. An integer (0..15) corresponding to CP/M drives A..P.

F. A 21-element integer array corresponding roughly to the CP/M FCB. F should be initialized to zeros. The order of the fields in the FCB is irrelevant, since routines are provided that can access the important fields.

NAME. A 14-element character array corresponding to a CP/M file name specification (with or without wild cards).

LOGVEC. A 16-element integer array with values (0 or 1), where values of 1 correspond to the drives that have been logged in.

ALLVEC. An integer array with values (0 or 1), where values of 1 correspond to blocks that have been allocated; the number of elements in the array correspond to the number of blocks per disk.

RODVEC. A 16-element integer array with values (0 or 1) where values of 1 correspond to the drives that have been write protected (read only drives).

USRNUM. An integer with values 0 through 15, 63 for "?", or 255 for "don't care".

FLAG. An integer from 1 to 11 corresponding to one of the CP/M attribute bitsei.

EXTENT. An integer with values 0 .. 255 or 63 for "?".

CRNUM. An integer corresponding to the desired current record number for a file being accessed sequentially.

RRNUM. An integer corresponding to the desired random record number for a file being accessed randomly.

B. An integer array large enough to contain the CP/M buffer (byte array); this local buffer is necessary since the CPM buffer cannot be directly accessed from a FORTRAN program.

BYTNUM. An integer from 1 to twice the local buffer size, which is the desired byte index into the CP/M buffer.

VALUE. An integer from 0 through 255 (i.e., a byte value).

G. A 21-element integer array corresponding roughly to a special CP/M FCB that is used during directory searches; G should be initialized to zeros.

EXAMPLE -- FORTRAN Program Using FCPM

```
$USES FCPM
      PROGRAM DEMO
      IMPLICIT INTEGER (C)
C
C  This program serves principally as a demonstration of how some of
C  the more useful routines in the FCPM unit are invoked.
C  At the same time, it accomplishes the following:
C     1)  copies the first 10 records of FILE1 to FILE2
C         using sequential reads and writes.
C     2)  closes and reopens FILE2 and computes its size so
C         that more records can be appended.
C     3)  sets the random record number for FILE1 to the current
C         record and copies the remainder of the records using
C         random reads and writes.
C  NOTE:     In order to simplify the demonstration, very few
C            error checks are included.
C  WARNING:  If the specified output file already exists, it is
C            deleted by this demonstration program.
C
      INTEGER RSLT,RRNUM,RRNUM2
```

```
      INTEGER F(21),F2(21)
      INTEGER B(512)
      CHARACTER*14 NAME
C  Initialize FCB arrays with zeros
      DO 1000 I = 1,21
      F(I) = 0
 1000 F2(I) = 0
      WRITE(*,100) 'Enter FILE1 name:
      READ(*,200) NAME
C  Set FILE1 name specification in the FCB
      RSLT = CPMSFN(F,NAME)
C  Open FILE1
      RSLT = CPMI(F,15)
      WRITE(*,100) 'Enter FILE2 name:
      READ(*,200) NAME
C  Set FILE2 name specification in the FCB
      RSLT = CPMSFN(F2,NAME)
C  Open FILE2 - delete it if it already exists
      RSLT = CPMI(F2,15)
      IF (RSLT .NE. 0) GOTO 20
C  Delete FILE2
      RSLT = CPMI(F2,19)
C  Make (create) FILE2
   20 RSLT = CPMI(F2,22)
C  Set current record numbers to zeros for FILE1 & FILE2
      CALL CPMSCR(F,0)
      CALL CPMSCR(F2,0)
      DO 2000 I = 1,10
C  Read record from FILE1 sequentially
      RSLT = CPMI(F,20)
      IF (RSLT .NE. 0) GOTO 40
C  Write record to FILE2 sequentially
 2000 RSLT = CPMI(F2,21)
C  Close and re-open FILE2
      RSLT = CPMI(F2,16)
      RSLT = CPMI(F2,15)
C  Compute size of FILE2
      RSLT = CPMI(F2,35)
C  Save random record number for FILE2
      RRNUM2 = CPMGRR(F2)
C  Set random record number for FILE1 to current record
      RSLT = CPMI(F,36)
C  Save random record number for FILE1
      RRNUM = CPMGRR(F)
C  Read record from FILE1 randomly
   30 RSLT = CPMI(F,33)
```

```
      IF (RSLT .NE. 0) GOTO 40
C  Get CPM buffer
      CALL CPM26G(B)
C  At this point the buffer could be examined and modified
C  Put CPM buffer
      CALL CPM26P(B)
C  Write record to FILE2 randomly
      RSLT = CPMI(F2,34)
      RRNUM = RRNUM + 1
      RRNUM2 = RRNUM2 + 1
C  Set random record numbers for FILE1 & FILE2 to next records
      CALL CPMSRR(F,RRNUM)
      CALL CPMSRR(F2,RRNUM2)
      GOTO 30
C  Close FILE1 & FILE2
   40 RSLT = CPMI(F,16)
      RSLT = CPMI(F2,16)
  100 FORMAT(A )
  150 FORMAT(A)
  200 FORMAT(A14)
  999 END
```

III.3.9 BASIC Interface to CP/M XenoFile

This section explains how p-System BASIC programs access CP/M disks. The user must have read and understood the preceding information in this manual as well as the CP/M Interface Guide.

A BASIC program invokes XenoFile by using the unit BCPM. This unit should be installed in *SYSTEM.LIBRARY so that it can be found during compilation and execution. The BASIC program must then include the directive "USES BCPM" in order to use the routines provided by BCPM.

NOTE: BCPM expects BASIC programs to pass only element one of an integer array argument.

The following listing is an example of correspondence between CP/M BDOS calls 12 through 36 and BCPM callsx.

Call #	BASIC p-System sequence	Description
12	RSLT = CPM12()	Return the version number
13	RSCALL CPM13	Reset the disk system
14	RSLT = CPM14(DRIVE)	Select disk
15	RSLT = CPMI(F(1),15)	Open file
16	RSLT = CPMI(F(1),16)	Close file
17	RSLT = CPMI(F(1),17)	Search for first
18	RSLT = CPMI(F(1),18)	Search for next
19	RSLT = CPMI(F(1),19)	Delete file
20	RSLT = CPMI(F(1),20)	Read a file sequentially
21	RSLT = CPMI(F(1),21)	Write a file sequentially
22	RSLT = CPMI(F(1),22)	Make a new file
23	RSLT = CPM23(F(1),NAME$)	Rename file
24	CALL CPM24(LOGVEC(1))	Return login vector
25	RSLT = CPM25()	Return current disk
26	CALL CPM26G(B(1))	Get CPM buffer
	CALL CPM26P(B(1))	Put CPM buffer
27	CALL CPM27(ALLVEC(1))	Get allocation vector
28	CALL CPM28(DRIVE)	Write protect disk
29	CALL CPM29(RODVEC(1))	Get read-only vector
30	RSLT = CPMI(F(1),30)	Set file attributes
31	NOT SUPPORTED	
32	RSLT = CPM32G()	Get user code
	CALL CPM32S(USRNUM)	Set user code
33	RSLT = CPMI(F(1),33)	Read random
34	RSLT = CPMI(F(1),34)	Write random
35	RSLT = CPMI(F(1),35)	Compute file size
36	RSLT = CPMI(F(1),36)	Set random record

The following additional BASIC calls with no corresponding CP/M BDOS function calls are also available in BCPM:

```
CALL CPMSFN(F(1),NAME$)          Set file name
CALL CPMGFN(F(1),NAME$)          Get file name
RSLT = CPMGFL(F(1),FLAG)         Get attribute flag
CALL CPMSFL(F(1),FLAG)           Set attribute flag to 1
CALL CPMCFL(F(1),FLAG)           Clear attribute flag to 0
CALL CPMFCB(G(1))                Get FCB for calls 17 & 18
RSLT =  CPMGE(F(1))              Get file extent number
CALL CPMSE(F(1),EXTENT)          Set file extent number
RSLT = CPMGCR(F(1))              Get current record number
CALL CPMSCR(F(1),CRNUM)          Set current record number
RSLT = CPMGRR(F(1))              Get random record number
CALL CPMSRR(F(1),RRNUM)          Set random record number
RSLT = CPMGB(B(1),BYTNUM         Get byte out of buffer
                                 (Used after calling CPM26G)
CALL CPMSB(B(1),BYTNUM,VALUE)    Set byte in buffer
                                 (Used before calling CPM26P)
```

The following paragraphs define the arguments to the preceding calls.

RSLT. An integer value returned by the function.

DRIVE. An integer (0..15) corresponding to CP/M drives A..P.

F. A 21-element integer array corresponding roughly to the CP/M FCB. F should be initialized to zeros. The order of the fields in the FCB is irrelevant, since routines are provided that can access the important fields.

NAME. A 14-element character array corresponding to a CP/M file name specification (with or without wild cards).

LOGVEC. A 16-element integer array with values (0 or 1), where values of 1 correspond to the drives that have been logged in.

ALLVEC. An integer array with values (0 or 1), where values of 1 correspond to blocks that have been allocated; the number of elements in the array correspond to the number of blocks per disk.

RODVEC. A 16-element integer array with values (0 or 1) where values of 1 correspond to the drives that have been write protected (read only drives).

USRNUM. An integer with values 0 through 15, 63 for "?", or 255 for "don't care".

FLAG. An integer from 1 to 11 correspo

EXTENT. An integer with values 0 .. 255 or 63 for "?".

CRNUM. An integer corresponding to the desired current record number for a file being accessed sequentially.

RRNUM. An integer corresponding to the desired random record number for a file being accessed randomly.

B. An integer array large enough to contain the CP/M buffer (byte array); this local buffer is necessary since the CPM buffer cannot be directly accessed from a FORTRAN program.

BYTNUM. An integer from 1 to twice the local buffer size, which is the desired byte index into the CP/M buffer.

VALUE. An integer from 0 through 255 (i.e., a byte value).

G. A 21-element integer array corresponding roughly to a special CP/M FCB that is used during directory searches; G should be initialized to zeros.

Example BASIC Program Using BCPM

```
USES BCPM
REM
REM This program serves only as a demonstration of how some of
REM the more useful routines in the BCPM unit are invoked.
REM At the same time, it accomplishes the following:
REM     1)  copies the first 10 records of FILE1 to FILE2
REM         using sequential reads and writes.
REM     2)  closes and reopens FILE2 and computes its size so
REM         that more records can be appended.
REM     3)  sets the random record number for FILE1 to the current
REM         record and copies the remainder of the records using
REM         random reads and writes.
REM NOTE:     In order to simplify the demonstration, very few
REM           error checks are included.
REM WARNING:  If the specified output file already exists, it is
REM           deleted by this demonstration program.
REM
       INTEGER RSLT,RRNUM,RRNUM2
       DIM INTEGER F(21)
       DIM INTEGER F2(21)
       DIM INTEGER B(512)
       DIM NAME$*14
REM Initialize FCB arrays with zeros
       FOR I = 1 TO 21 :: F(I) = 0 :: F2(I) = 0 :: NEXT I
       ACCEPT "Enter FILE1 name: ":NAME$
REM Set FILE1 name specification in the FCB
       RSLT = CPMSFN(F(1),NAME$)
REM Open FILE1
       RSLT = CPMI(F(1),15)
       ACCEPT "Enter FILE2 name: ":NAME$
REM Set FILE2 name specification in the FCB
       RSLT = CPMSFN(F2(1),NAME$)
REM Open FILE2 - delete it if it already exists
       RSLT = CPMI(F2(1),15)
       IF RSLT <> 0 THEN GOTO 20
REM Delete FILE2
       RSLT = CPMI(F2(1),19)
REM Make (create) FILE2
   20 RSLT = CPMI(F2(1),22)
REM Set current record numbers to zeros for FILE1 & FILE2
       CALL CPMSCR(F(1),0)
       CALL CPMSCR(F2(1),0)
       FOR I = 1 TO 10
REM Read record from FILE1 sequentially
```

```
      RSLT = CPMI(F(1),20)
      IF RSLT <> 0 THEN GOTO 40
REM Write record to FILE2 sequentially
      RSLT = CPMI(F2(1),21)
      NEXT I
REM Close and re-open FILE2
      RSLT = CPMI(F2(1),16)
      RSLT = CPMI(F2(1),15)
REM Compute size of FILE2
      RSLT = CPMI(F2(1),35)
REM Save random record number for FILE2
      RRNUM2 = CPMGRR(F2(1))
REM Set random record number for FILE1 to current record
      RSLT = CPMI(F(1),36)
REM Save random record number for FILE1
      RRNUM = CPMGRR(F(1))
REM Read record from FILE1 randomly
   30 RSLT = CPMI(F(1),33)
      IF RSLT <> 0 THEN GOTO 40
REM Get CPM buffer
      CALL CPM26G(B(1))
REM At this point the buffer could be examined and modified
REM Put CPM buffer
      CALL CPM26P(B(1))
REM Write record to FILE2 randomly
      RSLT = CPMI(F2(1),34)
      RRNUM = RRNUM + 1
      RRNUM2 = RRNUM2 + 1
REM Set random record numbers for FILE1 & FILE2 to next records
      CALL CPMSRR(F(1),RRNUM)
      CALL CPMSRR(F2(1),RRNUM2)
      GOTO 30
REM Close FILE1 & FILE2
   40 RSLT = CPMI(F(1),16)
      RSLT = CPMI(F2(1),16)
      END
```

III.4 Turtlegraphics

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

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

A pixel is a single picture element or point on the display.

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

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

The programmer may also define a viewport, or window on the display. This limits all graphic activity to within that port.

Turtlegraphics is shipped in two ways. If the p-System with turtlegraphics is adapted to a particular hardware configuration, then the graphic routines are already tailored to the display. The unit Turtlegraphics is already installed in *SYSTEM.LIBRARY, and a program may use its routines by including the following declaration.

```
USES turtlegraphics; (or an equivalent declaration in BASIC or FORTRAN).
```

If turtlegraphics is purchased as a separate, configurable product, then the user must write a number of assembly language routines that control the graphic display. These routines are called by the turtlegraphics unit and must be written and tested before turtlegraphics may be used.

Turtlegraphics and its capabilities are accessible to BASIC. There are no parameter type conflicts with BASIC. The only restriction is that BASIC must refer to all turtlegraphics identifiers by their first eight characters. For example, ReadPixel becomes ReadPixe.

A special, user-written unit is required to interface FORTRAN programs to turtlegraphics. The interface of this unit can contain only legal FORTRAN types, for example, integer, real, or alpha<digit..digit>. The unit should be structured to use turtlegraphics in its implementation section.

III.4.1 Using Turtlegraphics

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

For quick reference, the next-to-last subsection of Section III.4.1 contains a listing of the interface part of the turtlegraphics unit.

The last subsection of Section III.4.1 contains a sample program that illustrates a number of the turtlegraphics routines.

III.4.1.1 The Turtle

The turtle is an imaginary creature in the display screen that will draw lines as the user moves it around the display. The turtle can move in a straight line (Move), move to a particular point on the display (Moveto), turn relative to the current direction (Turn), and turn to a particular direction (Turnto).

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

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

NOTE: The turtle may be moved anywhere; it is not limited by the size of the figure or the size of the display. But, only movements within the figure will be visible.

To use the turtle in a figure other than the actual display, the programmer may call Activate_Turtle.

The following paragraphs describe the routines that control the turtle.

Procedure Move (distance: real);

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

Procedure Moveto (x,y: real);

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

Procedure Turn (rotation: real);

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

Procedure Turnto (heading: real);

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

Procedure Pen_color (shade: integer);

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

The color of the pen depends on the way the video display is set. If your turtlegraphics is already configured, the available colors are described in the documentation for your hardware. If you must configure turtlegraphics yourself, then the assembly language routines you write will control the display color: see Chapter IV of this supplement.

A sample set of colors might be:

0 = Wild card color
1 = Green
2 = Red
3 = Yellow

Turtlegraphics uses a numeric designation for color instead of a symbolic designation like the word blue or red to maintain the p-System language and hardware compatibility. For example, while Pascal would allow the use of symbolic color designations, BASIC and FORTRAN would not.

The term wild card refers to the standard background color of your display. This depends on your display hardware, and might be called a "hard" background (you may or may not be able to change it from a program: this depends on your hardware configuration). 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).

You may also use black and white graphics, in which case the colors might be simply:

0 = Black
1 = White

Procedure Pen_mode (mode: integer);

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

These are the possible modes:

0 = Nop - does not alter the figure.

1 = Substitute - writes the current pen color.

2 = Overwrite - writes the current pen color.

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

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

Values greater than 4 are treated as Nop.

These descriptions apply to movements of the turtle. They have a more complex meaning when a figure is copied onto a figure that is already displayed.

Function Turtle_x : real;

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

Function Turtle_y : real;

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

Function Turtle_angle : real;

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

Procedure Activate_Turtle (screen: integer);

Specifies to which figure subsequent turtlegraphics commands are directed. Each invocation of this procedure puts the previously active turtle to sleep and awakens the turtle in the designated figure. When turtlegraphics is initialized, the turtle in the actual display is awake. The initial position of the turtle is (0,0) or the bottom left comor of the screen, ready to move right.

III.4.1.2 The Display

We refer to the initial background of the display as the wild card color. The wild card color (color 0) depends on your hardware (or it may be possible for you to set it from a program). The default is typically 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: If you use turtlegraphics (or customized routines of your own) to alter the settings of your display, it is a good idea to reset everything before your program terminates. Usually it is not possible for the display to return to its original state, and the p-System software has no knowledge of what that original state was.

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

Fills the specified figure ("screen") with the specified color ("shade"). If screen = 0, which indicates the actual display screen, then only the current viewport is shaded. For user-created figures, the entire figure is shaded.

**Procedure Background (screen: integer;
shade: integer);**

Specifies the backgound color for a figure. The initial background color of all figures is the wild card color.

III.4.1.3 Labels

It is possible to draw legends, labels, and so forth on the display while using the turtlegraphics unit. This is done by calling either WChar or WString. The character or string appears at the location of the currently active turtle. The text is displayed in the type font defined by the file *SYSTEM.FONT (See Chapter IV of this supplement to find out how to define a font).

Procedure WChar (c: char; copymode, shade: integer);

Writes a single character at the position of the currently active turtle, using the indicated pen mode and color. The character is always displayed horizontally, regardless of the active turtle's direction.

Procedure WString (s: string; copymode, shade: integer);

Writes a string starting at the position of the currently active turtle, using the indicated pen mode and color. The string is always displayed horizontally, regardless of the active turtle's direction.

III.4.1.4 Scaling

When a programmer wishes to display data without altering the input data itself, it is possible to set scaling factors that translate data into locations on the display. This is done with Display_scale. The display scale applies globally to all figures.

Because of the shape of the actual display, data for particular shapes (especially curved figures) might become distorted when using a "straight" display scale. In this case, the function Aspect_ratio can be used to preserve the "squareness" of the figure.

Procedure Display_scale
(min_x,min_y,max_x,max_y: real);

Defines the range of input coordinate positions that are to be visible on the display. Turtlegraphics maps the user's coordinates into pixel locations according to the scale specified in Display_scale.

This procedure sets the viewport to encompass the whole display. The display bounds apply to input data. For the actual display, these bounds can be any values the user requires, but for user-created figures (0,0) is the lower left-hand corner.

If your turtlegraphics package is tailored to your hardware, then the default display scale is already supplied. If you purchased turtlegraphics as a separate, configurable product, then you must supply the parameters for your own display (these must be returned by the user-written procedure Query_Environment. (See Section III.4.1).

The following lines are an example of a default scale. It is simply the array of pixels on the FULL display.

```
min_x = 0, max_x = 319
min_y = 0, max_y = 199
```

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 following ratio is the same as the aspect ratio of the physical screen upon which the image is displayed.

```
(max_x - min_x) / (max_y - min_y)
```

When the turtlegraphics unit is initalized, min_x and min_y are set to 0. max_x is initialized to the number of pixels in the x direction, and max_y is initialized to the number of pixels in the y direction. In order to change to different units that still have the same aspect ratio, a call similar to the following example can be used.

```
Display_scale(0, 0, 100*ASPECT_RATIO, 100);
```

This utilizes Function Aspect_ratio described above, and makes the y axis 100 units long.

Turtlegraphics always treats the turtle as being in a fixed pixel location. Changing the scaling of the system with a call to this routine in the middle of a program does not alter the pixel position of any of the turtles in the figures. However, the values returned from X_pos and Y_pos may change.

III.4.1.5 Figures and the Port

The programmer can create and delete new figures, each with its own turtle. When a new figure is created, it is assigned an integer, and this integer refers to that figure in subsequent calls to turtlegraphics procedures. New figures can be saved (Put_Figure) or displayed on the screen (Getfigure).

The actual display is always referred to as figure 0.

The active portion of the display can be restricted by calling viewport, which creates a "window" on the screen in which all subsequent graphics activity takes place. The user might create a figure, specify the port, then display that figure (or a portion of it) within the port. Specifying a viewport does not restrict turtle activity, it merely restricts what is displayed on the screen.

User-created figures can be saved in p-System disk files.

Function Create_Figure (x_size,y_size: real): integer

Creates a new figure that is rectangular, and has the dimensions (x_size, y_size), where (0,0) designates the lower left-hand corner. The dimensions are in units of the current display scale. The figure is identified by the integer returned by Create_figure.

When a figure is created it contains its own turtle, which is located at the initialization position or 0,0 and has a direction of 0 (it faces the right-hand side of the figure). The turtle in a user-created figure can be used by calling Activate_Turtle.

Procedure Delete_figure (screen: integer);

Discards a previously created display figure area.

Though figures may be created and destroyed, indiscriminate use of these contructs may rapidly exhaust the memory available in the p-System due to heap fragmentation. For example, a figure may be created using Create_Figure (or it may be read in from disk using Function Load_Figure, described below). If possible, after that figure is used (for example, with a Get_Figure, Put_Figure, Load_figure or Store_Figure operation) it should be deleted before other figures are created. If many figures are created, and randomly deleted, the heap fragmentation problem may occur.

**Procedure Get_Figure (source_screen: integer;
corner_x,corner_y: real; mode: integer);**

Transfers a user-created figure (the source) to the display screen (the destination) using the drawing mode specified. The figure is placed on the display such that its lower left corner is at (corner_x, corner_y). The x and y positions are specified in the units of the current display scale. If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

The following items define the drawing mode numbers.

0 Nop. Does not alter the destination.

1 Substitute. Each pixel in the source replaces the corresponding pixel in the destination.

2 Overwrite. Each pixel in the source that is not of the source's background color replaces the corresponding pixel in the destination.

3 Underwrite. Each pixel in the source that is not of the source's background color is copied to the corresponding pixel in the destination only if the corresponding pixel is of the destination's background color.

4 Complement. For each pixel in the source that is not of the source's background color, the corresponding pixel in the destination is complemented.

Values greater than 4 are treated as Nop.

If a portion of the source figure falls outside the display or the window, it is set to the source's background color.

Procedure Put_Figure (destination_screen: integer; corner_x,corner_y: real; mode: integer);

Transfers a portion of the display screen to a user-created figure using the drawing mode specified (see above). The portion transferred to the figure is the area of the display that the figure covers when it is placed on the display with its lower left-hand corner is at (corner_x, corner_y). If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

NOTE: When a figure is moved to the display by Get_Figure, further modifications to the display do not affect the copy of the figure that is saved in memory. If the user wishes to save the results of graphics work on the display, it is necessary to call Put_Figure.

Procedure Viewport (min_x,min_y, max_x,max_y: integer);

Defines the boundaries of a "window" that confines subsequent graphics activities. The viewport procedure applies only to the actual display. When a window has been defined, graphics activities outside of it are neither displayed nor retained in any way. Therefore, lines, or portions thereof, that are drawn outside the window are essentially lost and will not be displayed (this is true even if the window is subsequently expanded to encompass a previously drawn line). The viewport boundaries are specified in the units of the current display scale. If the specified size of the viewport is larger than the current range of the display, the Viewport is truncated to the display limits.

III.4.1.6 Pixels

It is possible to ascertain (Read_pixel) or alter (Set_pixel) the color of an individual pixel within a given figure. These routines are more specific than the turtle-moving routines. They are less straightforward to use, but give the programmer greater control.

Function Read_pixel (screen: integer; x,y: real): integer;

Returns the value of the color of the pixel at the x,y location in the specified figure. The x,y location is specified in the units of the current Display_scale.

Procedure Set_pixel (screen: integer; x,y: real; shade: integer);

Sets the pixel at the x,y location of the specified figure to the specified color. The x,y location is specified in the units of the current Display_scale.

III.4.1.7 Fotofiles

The programmer may create disk files that contain turtlegraphics figures. New figures may be written to a file, and old figures restored for viewing or modification.

When figures are written to a file, they are written sequentially, and assigned an 'index' that is their location in the file. They may be retrieved "randomly" by using this index value.

The p-System name for files of figures always contains the suffix '.FOTO'. It is not necessary to use this suffix when calling Read_figure_file or Write_figure_file (if absent, it will be supplied automatically).

Function Read_figure_file (title: string): integer;

Specifies the title of a file from which all subsequent figures will be loaded. If a figure file is already open for reading when this function is called, it is closed before the new file is opened. Only one figure file may be open for reading at a single time. This function returns an integer value which is the ioresult of opening the file.

Function Write_figure_file (title: string): integer;

Creates an output file into which user-created figures may be stored. If another figure file is open for writing when this function is called, it is closed, with lock, before the new file is created. Only one figure file may be open for writing at a single time. This function returns an integer result which is the ioresult of the file creation.

Function Load_figure (index: integer): integer;

Loads the indexed figure from the current input figure file and assigns it a new, unique, figure number. An automatic Create_figure is performed. If the operation fails for any reason, a Figure_number of zero (0) is returned.

Function Store_figure (figure: integer): integer;

Sequentially writes the designated figure to the output figure file. The function returns an integer that is the figure's positional index in the current output figure file. Positional indexes start at one (1). If the index returned equals zero (0), turtlegraphics did not successfully store the figure.

III.4.1.8 Routine Parameters

The next page shows the interface section for the turtlegraphics unit, including the parameters to all turtlegraphics routines:

```
unit turtlegraphics;

    interface

    procedure Display_scale( min_x, min_y,
                             max_x, max_y: real);
    function Aspect_ratio : real;
    function Create_figure( x_size, y_size:
                             real ) : integer;
    procedure Delete_figure( screen:
                             integer );
    procedure Viewport( min_x, min_y, max_x,
                             max_y : real );
    procedure Fillscreen( screen:
                             integer; shade:
                             integer);
    procedure Background( screen: integer;
                             shade : integer );
    function Read_pixel( screen: integer;
                             x, y : real ) :integer;
    procedure Set_pixel( screen:integer;
                             x,y:real; shade:color );
    procedure Get_Figure( source_screen:
                             integer,
                             corner_x, corner_y: real;
                             mode : integer );
    procedure Put_Figure( destination_screen:
                             integer,
                             corner_x, corner_y: real;
                             mode : integer );
    function Read_figure_file( title : string ):
                             integer;
    function Write_figure_file( title : string ):
                             integer;
    function Load_figure( index : integer ):
                             integer;
    function Store_figure( figure: integer):
                             integer;
    procedure Activate_Turtle( screen:
                             integer );
    function Turtle_x : real;
    function Turtle_y : real;
    function Turtle_angle : real;
    procedure Move( distance : real );
    procedure Moveto( x,y : real );
    procedure Turn( rotation : real );
```

```
procedure Turnto(  heading : real );
procedure Pen_mode( mode : integer );
procedure Pen_color( shade : integer );
procedure WChar( c: char; copymode, shade: integer );
procedure WString( s: string; copymode, shade: integer);
```

III.4.1.9 Sample Program

Here is a sample program that illustrates a number of turtlegraphics routines:

```
program Spiraldemo;

  uses Turtlegraphics;

  const  nop = 0;
         substitute = 1;

  var I, J, Mode: integer;
      C: char;
      Color: integer;
      Seed: integer;
      LX, LY, UX, UY: real;

  function Random (Range: integer): integer;
    begin
      Seed:= Seed * 233 + 113;
      Random:= Seed mod Range;
      Seed:= Seed mod 256;
    end;

  procedure ClearBottom;
    {clears bottom line of screen
     for prompts}
    begin
      Penmode (nop);
      Moveto (0, 0);
      WString ('                                        ', substitute, 1);
    end;

  begin
    ClearBottom;        {various initializations}
    WString ('ENTER RANDOM NUMBER: ', substitute, 1);
    read(keyboard, Seed);
    ClearBottom;
    Display_Scale (0, 0, 200*Aspect_Ratio, 200);
```

```
    {Aspect_Ratio used so
     pattern will be round}
Color:= 0;
WString ('ENTER VIEWPORT LL CORNER: ', substitute, 1);
read(keyboard, LX,LY);
ClearBottom;
WString ('ENTER VIEWPORT UR CORNER: ', substitute, 1);
read(keyboard, UX,UY);
ClearBottom;
WString ('PENMODE = ', substitute, 1);
read(keyboard, MODE);

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(90)+90;
    {angle by which turtle will move
     note that turtle begins facing right
     and will move counterclockwise
     (J is positive)}

for I:= 2 to 200 do
    {draw spiral in 200 segments
     of increasing length}
    begin
          {cycle through 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}
end.
```

IV. INSTALLATION GUIDE SUPPLEMENT

This chapter covers information that you will need to know if you have purchased the UCSD p-System as an adaptable system, or if you are going to upgrade a running version IV adaptable system to the current release level. It describes how the installation of adaptable systems differs from the description given in the UCSD p-System Installation Guide. The sections that are pertinent to your particular system should be read before reading the Installation Guide.

Several other miscellaneous topics related to installing and configuring the p-System are covered.

NOTE: Only the extended SBIOS is supported by the current release. If you wish to use a non-extended SBIOS you must supply stub routines (which pop parameters and return) for the extended SBIOS routines.

If you are only going to upgrade a running version IV adaptable system, you will need to understand what is described in the following sections:

IV.1.1 Release Disk Format
IV.1.3 Required New SBIOS Routines: Quiet and Enable
IV.1.4 New Definition for PRNSTAT and SETTRAK
IV.1.6 Installing GOTOXY

If you wish to install print spooling, you will need the information in the following sections:

IV.1.2 Installing Events
IV.1.4 New Definition for PRNSTAT and SETTRAK
IV.3 Setup

If you have an American National Standards Institue (ANSI) terminal, or if you plan to library or memlock the OS unit SCREENOPS (regardless of the type of terminal you have), or if you wish to accept three character code sequences from the keyboard, you should read section:

IV.1.7 p-System support for ANSI Terminals

The installation of the 8086 adaptable system is described in section:

IV.2 The 8086 Adaptable System

There are several new fields within SYSTEM.MISCINFO in the current release. They are related to extended memory, print spooling, subsidiary volumes, etc. These fields are set using the utility SETUP described in section:

IV.3 Setup

In order to install an adaptable TURTLEGRAPHICS package, you should read:

IV.4 Installing TURTLEGRAPHICS

Finally, listings of the new adaptable system release disks are given for each processor in section:

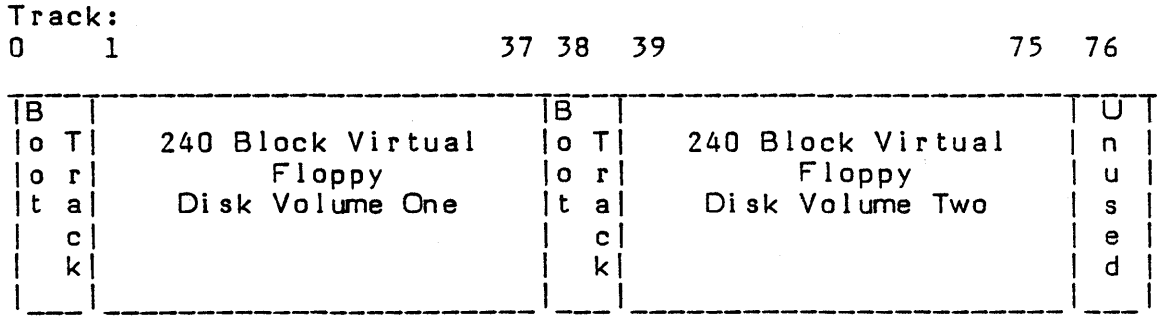
IV.5 Adaptable System Release Disk Directories

IV.1 Installation of Adaptable Systems

This section gives information which is pertinent to all adaptable systems with the current p-System release.

IV.1.1 Release Disk Format

The release disks for the current adaptable systems are formatted into virtual floppies, as with previous adaptable systems. However, there are now two (rather than three) virtual floppies per disk. Each virtual floppy contains 240 blocks. The following diagram illustrates the disk images of the virtual floppies:



NOTE: Remember to back up the disks that you have received with the p-System before you do anything with them.

IV.1.2 Installing Events

Events are discussed in Chapter II of this supplement. The SBIOS routines QUIET and ENABLE (discussed in Section IV.1.3), and the BIOS routine EVENT (discussed in this section) are pertinent to the new event handling facilities.

EVENT is a BIOS routine that may be called from the SBIOS. When an SBIOS routine detects an event such as a hardware interrupt or a key pressed on the console's keyboard, it may call EVENT with an appropriate event number. This event number may be associated with a semaphore in a high-level language (in UCSD Pascal, this is accomplished by the ATTACH intrinsic).

The event numbers 0..31 are reserved for the p-System's use, and the event numbers 32..63 are available for user definition. The events that you may choose to signal, and what you choose to do with them, are entirely up to you.

The following table gives machine-specific information concerning calls to EVENT:

Processor	BIOS vector offset	Parameter passed in register
Z80	6	HL
8080	6	HL
8086	4	DI
6502	6	on stack, (SP)+2 and (SP)+3

EVENT will destroy the information in all of the microprocessor's working registers. The SBIOS is responsible for saving all the necessary processor state prior to calling EVENT and restoring it upon return.

IV.1.3 Required New SBIOS routines: QUIET and ENABLE

This section describes the two new SBIOS routines that you must provide in order to bring up the current p-System. They are called QUIET and ENABLE.

QUIET must disable any P-machine events from occurring. After a call to QUIET, no part of the SBIOS should call EVENT until the corresponding call to ENABLE has been made. The simplest way to do this may be to disable all processor interrupts. If your hardware configuration does not allow you to do this, you must devise some other scheme for disabling events. A software mechanism may be required to implement QUIET. If you wish, you may write code to queue events during times when QUIET is in effect.

ENABLE allows P-machine events to occur. This may be done by simply re-enabling processor interrupts, or by a scheme that corresponds to the one used by QUIET.

QUIET and ENABLE are always called in pairs by the p-System. These calls are never nested.

The following table gives the machine-specific SBIOS vector offsets for QUIET and ENABLE:

Processor	QUIET	ENABLE
Z80	54 hex	57 hex
8080	54 hex	57 hex
8086	38 hex	3A hex
6502	54 hex	57 hex

These vector offsets follow the offsets for the extended SBIOS routines. The extended SBIOS routines are not required to be fully implemented (they may be stub routines which pop the parameters and return, signalling offline), but QUIET and ENABLE must be implemented.

IV.1.4 New Definition for PRNSTAT and SETTRAK

Definitions for the two SBIOS routines PRNSTAT and SETTRAK have been modified for this release. PRNSTAT must be upgraded in order to run the current p-System.

The SBIOS routine PRNSTAT (described in the Installation Guide) now has an additional parameter. This parameter is an I/O direction flag.

When the I/O direction flag is non-zero, it indicates input and PRNSTAT should behave as in previous versions.

When the I/O direction flag is 0, it indicates output and the following information should be returned by PRNSTAT:

The online/offline status of the printer is returned as the first parameter in the same manner as before.

The second parameter returns:
0 if the printer is not busy (i.e. can accept a character)
FF if the printer is busy

The new I/O direction flag is passed as follows:

Processor	I/O Direction Parameter Passed in Register
Z80	C
8080	C
8086	AX
6502	A

As long as the printer is busy the print spooler will not send any characters.

NOTE: This routine must be upgraded for print spooling to work.

The SETTRAK routine is compatible with earlier versions. It should be written, however, to accept a 16 bit track number if that is possible (i.e. if you are using disks that have more than 256 tracks).

On 8086 systems, the track number is passed in AX (as described in Section V.2.2).

On 8080 and Z80 systems, the track number is passed in BC. Previously, this parameter was passed only in register C. If you only require 8 bits to specify track numbers, you may ignore register B which contains the most significant byte.

On 6502 systems, this parameter is passed in XA. Previously, this parameter was passed only in register A. If you only require 8 bits to specify track numbers, you may ignore register X which contains the most significant byte.

IV.1.5 Floating Point Number Configuration

The p-System can be configured to use no real numbers, two word real numbers, or four word real numbers. If the floating point package is not used, more memory and disk space is available for other purposes. Two word reals has been the standard in the past. Four word (64 bit) real number representation significantly improves the accuracy and power of floating point operations.

NOTE: SofTech Microsystems expects to discontinue support of the two word floating point packages sometime in the future.

There are three components of the p-System which must be configured consistently with respect to real number size. These are the Operating System, the Interpreter, and the Compiler.

The Operating System is shipped with no REALOPS unit within it. If floating point operations are to be performed, a REALOPS unit must be librariedinto the OS using the Library utility (described in the Users' Manual). The four word and two word REALOPS units are found in two code files named:

REALOPS.xx.CODE

The "xx" will vary for different types of floating point representations, but will always contain a "4" or a "2". For example:

REALOPS.2.CODE
REALOPS.Z2.CODE

REALOPS.4.CODE

Choose the REALOPS unit with the desired real size (2 or 4), and library it into the Operating System (named SYSTEM.PASCAL on the disk). Be sure that KERNEL occupies slot 0 within the output file, and that USERPROG occupies slot 15. Rename the output file SYSTEM.PASCAL and place it on a boot disk.

NOTE: In addition to the general description of Library given in the Users' Manual, there is also an example of installing GOTOXY into the OS using Library that is

given in the Installation Guide. This example may be useful to read if you are not familiar with the Library utility. It does not matter what slot REALOPS occupies, as long as KERNEL occupies slot 0 and USERPROG occupies slot 15.

The Interpreter is also shipped with no real number facilities bound into it. A new Interpreter must be linked together with the appropriate floating point package if real numbers are to be used. This is done using the Linker as described in the Installation Guide under "Reconfiguring the Interpreter". The files that you must link together are slightly different, however.

On Z80 and 8080 systems you must choose from the following files when linking an interpreter together:

INTERP.8.CODE	8080 interpreter
INTERP.Z.CODE	Z80 interpreter
FP0.CODE	no reals
FP2.8.CODE	2 word reals for 8080
FP2.Z.CODE	2 word reals for Z80
FP4.CODE	4 word reals for either
RSP.CODE	run time support package
BIOS.CODE	BIOS choices as described
BIOS.C.CODE	in the Installation Guide
BIOS.CR.CODE	
BIOS.CRP.CODE	
INTER.X.CODE	extended SBIOS interface
INTER.CPM1.CODE	CP/M SBIOS interface, 1 disk drive
INTER.CPM2.CODE	CP/M SBIOS interface, 2 disk drives
INTER.CPM4.CODE	CP/M SBIOS interface, 4 disk drives
TERTBOOT.CODE	tertiary bootstrap

You must link exactly one file from each of the six groups just listed. For example, to create a two word floating point Z80 interpreter with full BIOS capabilities, you would choose the following files:

INTERP.Z.CODE
FP2.Z.CODE
RSP.CODE
BIOS.CRP.CODE
INTER.X.CODE
TERTBOOT.CODE

On 8086 adaptable systems, you will need to choose the files that make up the interpreter from a similar group, as described in Section IV.2 of this supplement.

On 6502 systems the files that are linked together are the same as before, except that there is only one Interpreter code file (INTERP.CODE) and one of the following floating point packages must be included:

FP0.CODE	no reals
FP2.CODE	two word reals
FP4.CODE	four word reals

The Pascal Compiler will default to the real size of the interpreter that it is running on. (This default may be overriden by the $R2 and $R4 Compiler options, however.)

The BASIC and FORTRAN compilers are both shipped in two different versions; one for each real size. Also, two versions of the corresponding runtime libraries are shipped.

IV.1.6 Installing GOTOXY

The standard GOTOXY contained in the Operating System runs on all machines. You will probably want to write and install into the OS your own GOTOXY because a machine-specific version of it will run faster than the generalized version. This is described in the Installation Guide.

In the current version, the variables CUR_X and CUR_Y within the SCREENOPS unit of the OS are now updated by GOTOXY. This allows the p-System to keep track of the cursor position on the screen. You should make sure that this is done by your machine-specific version of GOTOXY. The following sample GOTOXY updates the sample given in the Installation Guide.

```
{$U-}               { ALWAYS include this compiler directive. }
UNIT GOTOXY;
USES {$U SCREENOPS.CODE} SCREENOPS (SC_TX_PORT, SC_PORT);
INTERFACE
  PROCEDURE AGOTOXY(X,Y: INTEGER);

IMPLEMENTATION
  PROCEDURE AGOTOXY;

  CONST  TELL_LENGTH_MINUS_1 = 3,
           OFFSET = 32;
  { You may have to change these, depending on your terminal. }

  VAR     TELL: PACKED ARRAY [0..TELL_LENGTH_MINUS_1]
                  OF 0..255;
  BEGIN
    IF X>79 THEN X:=79
      ELSE IF X<0 THEN X:=0;
    IF Y>23 THEN Y:=23
      ELSE IF Y<0 THEN Y:=0;
    {  This range-checking is necessary.  The actual
       screenwidth and height may be different for you. }

    WITH SC_PORT DO
      BEGIN
        CUR_X:=X-COL;
        CUR_Y:=Y-ROW;
      END;

    { These first elements of TELL must contain
      the characters which tell your terminal to
      position the cursor at (X,Y):
      fill in the blanks...          }
    TELL[0] := ____;
    TELL[1] := ____;
    ...
    { The actual X and Y values are usually the
      last things in the array;
      the order may be different on your terminal. }
    TELL[TELL_LENGTH_MINUS_1 - 1]   := Y+OFFSET;
    TELL[TELL_LENGTH_MINUS_1]       := X+OFFSET;

    UNITWRITE(1,TELL,TELL_LENGTH_MINUS_1 + 1)
  END {AGOTOXY};
END {UNIT GOTOXY}.
```

IV.1.7 p-System Support For ANSI Terminals

American National Standard Institute (ANSI) terminals, such as the DEC VT100, are now supported by the p-System. There is a special ANSI SCREENOPS unit that must be installed into the operating system if you wish to use such a terminal. The unit is described here. Also, some related notes about keyboard input handling and the SETUP utility are covered.

NOTE: Even if you do not have an ANSI terminal, if you have any plans to use the utility Library to move SCREENOPS or if you plan to write programs which memlock the SCREENOPS unit, you should also read this section.

The standard SCREENOPS unit is installed in the OS as it is delivered. ANSI.CODE contains the ANSI SCREENOPS unit. In order to install it into the OS, you should use the utility Library as described in the Users' Manual.

(In addition to the general description of Library given in the Users' Manual, there is also an example of installing GOTOXY into the OS using Library that is given in the Installation Guide. This example may be useful to read if you are not familiar with the Library utility. That example incorrectly states that GOTOXY must occupy a slot greater than 15. Neither GOTOXY nor the SCREENOPS segments need to occupy any particular slot. However, KERNEL must occupy slot 0, and USERPROG must occupy slot 15.)

Both the standard SCREENOPS unit and the ANSI SCREENOPS unit have the same interface section, and are both divided into the following 4 segments:

```
SCREENOP
SEGSCINI
SEGSCPRO
SEGSCCHE
```

Whenever you are planning to move the SCREENOPS unit using Library, you must move all four of these segments. (Formerly, only the first two existed.) You must Library all four of these units into the OS when installing ANSI SCREENOPS, for instance.

As an improtant side note, if you plan to memlock SCREENOPS from a program, you should specify all three of the following segments:

```
SCREENOP
SEGSCPRO
SEGSCCHE
```

(SEGSINI is an initialization segment which is only called by the OS during system initialization. It does not need to be memlocked by user programs because it is never used during program execution.)

The ANSI SCREENOPS unit causes the p-System to ignore all of the syscom screen parameters which are set using the SETUP utility. This is because ANSI terminals all use standard screen codes. The ignored fields include:

BACKSPACE
ERASE LINE
ERASE SCREEN
ERASE TO END OF LINE
ERASE TO END OF SCRN
LEAD IN TO SCREEN
MOVE CURSOR HOME
MOVE CURSOR RIGHT
MOVE CURSOR UP

Also, the p-System may now accept sequences of three codes for keyboard input. The first code being the prefix, the second code being ignored, and the third code determining what key was typed. In SETUP, you would specify that the key is prefixed, and you would specify the third code as the key.

Two code sequences may still be recognized along with three code sequences. You must be careful in setting up which keys your system will recognize, however, so that there are no conflicts.

As an example, you might have special function keys which return three code sequences, and arrow keys which return two code sequences. You may specify, using SETUP, that a function key has significance as a p-System key (e.g. editor accept key) as long as the second code in the sequence does not match any other prefixed key according to SETUP.

If, for example a terminal returns codes as follows:

Left Arrow	<esc> a
Right Arrow	<esc> b
Up Arrow	<esc> c
Down Arrow	<esc> d
F1	<esc> [1
F2	<esc> [2

.
.
.

there would be no problem. When the p-System receives a prefix character (in

this case <esc>), it determines whether the next code matches a prefixed key. If it does not, then the p-System determines if the next code input matches a prefixed key. (If it does not, then the input is considered invalid.) In this case, typing a function key returns "[" as the second character; this does not conflict with a, b, c, or d.

The situation would be different if a keyboard were to return code sequences as follows:

```
Left Arrow      <esc> a
Right Arrow     <esc> b
Up Arrow        <esc> c
Down Arrow      <esc> d
F1              <esc> a 1
F2              <esc> a 2
    .
    .
    .
```

In this case, the F1 key could not be used as a p-System key because it conflicts with the left arrow.

NOTE: ANSI terminals can asynchronously send XON/XOFF sequences to delay transmission when they are being overrun. They do this using the following definitions:

```
XOFF = CTRL-S
XON  = CTRL-Q
```

CTRL-S is normally the p-System soft toggle for START/STOP (which does the identical task as the ANSI predefined CTRL-S and CTRL-Q). It confuses the p-System, however, if the terminal itself sends these same sequences.

If you have purchased the UCSD p-System as an adaptable system and you have an ANSI terminal, it is recomended that KEY FOR STOP be set to some value other than CTRL-S (using the SETUP utility). You should also implement XON/XOFF support in your console I/O routines.

IV.2 The 8086 Adaptable System

The p-System will now operate on the 8086 processor. The 8086 Adaptable System is not described in the Installation Guide.

This section describes the bootstraps, SBIOS interface, and BIOS interface for the 8086. You should already be familiar with the adaptable system in general, as

described in the Installation Guide.

The user is responsible for supplying SBIOS routines, as described in the Installation Guide and this supplement.

8086 systems that run version IV can take advantage of extended memory. Extended memory is described in Chapter 3 of this supplement.

The P-machine emulator ("interpreter") for the 8086 is broken into several modules. These are:

1) The main part of the Interpreter.

2) Floating point emulation.

3) The RSP (Runtime Support Package).

4) The BIOS (Basic I/O Subsystem).

5) The tertiary bootstrap routine.

The SYSTEM.INTERP supplied on the release disks contains no floating point support or I/O character queuing (INTERPX.CODE, FP0.CODE, RSP.CODE, BIOS.CODE, and TERTBOOT.CODE).

To link an 8086 Interpreter, the following modules (with their object code file names to the right) must be linked together:

Interpreter	INTERPX.CODE
Floating Point	FP0.CODE or FP2.CODE or FP4.CODE
Runtime Support Package	RSP.CODE
BIOS	BIOS.CODE or BIOS.C.CODE or BIOS.CR.CODE or BIOS.CRP.CODE
Tertiary Bootstrap	TERTBOOT.CODE

The "0", "2", and "4" in the FP codefiles distinguish between no real number support, two word, and four word floating point packages. FP0 must be linked in

if no real number support is chosen. The Operating System must contain routines that use the corresponding floating point size (REALOPS unit in SYSTEM.PASCAL) if real numbers are to be used.

The "C", "CR", and "CRP" in the BIOS code files indicate console, remote port, and printer character input queuing, as stated in the Installation Guide.

After linking, the utility program COMPRESS (described in the Users' Manual) must be run to form a memory image file of the Interpreter. The answers to the questions asked by COMPRESS are: NO relocatable output, base address = 0, the linker output file, and the desired filename (such as SYSTEM.INTERP).

IV.2.1 Bootstrapping the p-System

The steps necessary to get the Adaptable p-System started are described here. Please note that all address values are to be set relative to the CS register (i.e., the Interpreter load point specified by the user).

1. Load the primary bootstrap (Track 0, Sectors 1 and 2) and the user supplied SBIOS.

2. Set the Stack Pointer to any unused (but VALID) address value (such as C000). The SP register will be reset at the start of the TERTBOOT routine, so the value set here is not critical. Push the Adaptable System parameters onto the Stack as indicated in the Installation Guide. As previously stated, all address values are relative to the Interpreter load point. (See below.)

3. Set the CS register to the desired base (load point) of the Interpreter.

4. Do a short (intrasegment) jump to the bootstrap address (8000H -- CS-relative). (The CS register may also be set by making a long {intersegment} jump.) This invokes the primary bootstrap.

The primary bootstrap reads the secondary bootstrap from disk and jumps to it. The secondary bootstrap finds the Interpreter, reads it in, and jumps to the tertiary bootstrap (which is part of the Interpreter). Once the System is up and running, a user may wish to simplify the bootstrapping mechanism (see the Installation Guide).

IV.2.2 SBIOS Interface

The SBIOS routines are called from the BIOS through an address vector (NOT a jump vector, as with other processors). The following table briefly defines this interface. Addresses pointed to by the vector offsets are CS-relative.

Routine	Vector offset	Inputs	Outputs
SYSINIT	00	AX = pointer to Interpreter jump table	
SYSHALT	02		
CONINIT	04		AH = ioresult
CONSTAT	06		AH = ioresult AL = char present (0=False, FF=True)
CONREAD	08		AH = ioresult AL = char
CONWRIT	0A	AL = char	AH = ioresult
SETDISK	0C	AL = current disk	
SETTRAK	0E	AX = current track	
SETSECT	10	AL = current sector	

Routine	Vector offset	Inputs	Outputs
SETBUFR	12	AX = buffer addr (ES-relative)	
DSKREAD	14		AH = ioresult
DSKWRIT	16		AH = ioresult
DSKINIT	18		AH = ioresult
DSKSTRT	1A		
DSKSTOP	1C		
PRNINIT	1E		AH = ioresult
PRNSTAT	20	AX = control word	AH = ioresult AL = char present
PRNREAD	22		AH = ioresult AL = char
PRNWRIT	24	AL = char	AH = ioresult
REMINIT	26		AH = ioresult
REMSTAT	28		AH = ioresult AL = char present
REMREAD	2A		AH = ioresult AL = char
REMWRIT	2C	AL = char	AH = ioresult
USRINIT	2E	CL = unit #	AH = ioresult
USRSTAT	30	TOS(SP)->return i/o toggle ^statrec CL = device #	

Routine	Vector offset	Inputs	Outputs
USRREAD	32	TOS(SP)->return addr block # byte count ^buffer (DS-rel) device # control word	
USRWRIT	34	TOS(SP)->return addr block # byte count ^buffer (DS-rel) device # control word	
CLKREAD	36		AH = ioresult DX = high word CX = low word
QUIET	38		
ENABLE	3A		

Notes on the SBIOS:

1. The SBIOS must be assembled relative to zero (i.e., with no ORG directive), and relocated to its load address (relative to the CS or DS register) by using the COMPRESS utility.

2. The SBIOS is responsible for ensuring that the SS, DS, and CS registers are restored to the same state as at entry. All other registers are available for use.

3. The SBIOS vector contains only addresses, not jump instructions.

4. All SBIOS routines are CALLed indirectly from the BIOS through the vector. To return to the BIOS, a RET instruction is all that is needed (these calls and returns are short (intrasegment) CALLs and RETs).

5. The 8086 Adaptable System requires an Extended SBIOS (all of the above entry points). If user, remote, clock, or printers are not present, the routine should simply return an ioresult of 9 (offline).

6. QUIET and ENABLE are new entry points not documented in the Installation Guide. QUIET contains the functions necessary to disable P-machine "events" from occurring. ENABLE allows events to occur. (See Section IV.1 for more information on QUIET and ENABLE.) In most systems, the disable and enable interrupts functions (respectively) are all that are necessary (CLI and STI processor commands). A few hardware configurations may not permit the global disabling of processor interrupts, so some other scheme must be devised.

7. The routine vector passed to SYSINIT is described below.

IV.2.3 BIOS Routines Accessible to the SBIOS

The use of the new routine EVENT is described in Chapter II of this supplement, which discusses interrupt handling.

To use print spooling, use EVENT to signal a keys-ready interrupt (event number 19). See Chapter III of this supplement for more details.

Routine	Vector offset	Inputs
POLLUNITS	00	
DSKCHNG	02	BX = ^disk descriptor block (#tracks #sectors sector size interleaving skew first track)
EVENT	04	DI = event #

IV.2.4 The Primary Bootstrap

The primary bootstrap must be assembled relative to zero (i.e., without an ORG directive), and relocated to 8000H by using the utility COMPRESS (described in the Users' Manual).

The address of the Interpreter must be set to zero. The CS register must be set to the desired Interpreter base (shifted right by 4 bits). The bootstrapping and interpretation are relative to this CS value. The DS, ES, and SS registers will be set to the same value by the bootstrap. (ALL address values within the SBIOS must be relative to this CS value.)

The following parameters must be on the stack:

```
TOS --> SBIOS tester parameter  (ignored by primary boot)
        address of Interpreter  (CS-relative)
        address of SBIOS        (CS-relative)
        address of low word of contiguous memory  (CS-relative)
        address of high word of contiguous memory (CS-relative)
        number of tracks on boot disk
        number of sectors per track
        number of bytes per sector
        interleaving factor
        first interleaved track
        track-to-track skew
        maximum number of sectors in table
        maximum number of bytes per sector
```

The primary bootstrap must pop these values from the stack, load the SBIOS, and call the SBIOS SYSINIT routine. It must then read in the secondary bootstrap, which is located on Track 0.

Next, the primary bootstrap must push the following values onto the stack:

```
TOS --> address of Interpreter  (relative to CS)
        address of SBIOS        (CS-relative)
        address of low word of contiguous memory  (CS-relative)
        address of high word of contiguous memory (CS-relative)
        number of tracks on boot disk
        number of sectors per track
        number of bytes per sector
        interleaving factor
        first interleaved track
        track-to-track skew
        maximum number of sectors in table
        maximum number of bytes per sector
```

NOTE: These values are the same as before, except that the SBIOS test parameter is no longer present.

Finally, the primary bootstrap must jump to the secondary bootstrap.

IV.3 SETUP

The SETUP utility alters some new fields within SYSTEM.MISCINFO. These new fields are described in this section. The use of SETUP itself, and descriptions of the fields within SYSTEM.MISCINFO that were already alterable by the utility, are covered in the Installation Guide. The current version of SETUP is compatible

with earlier versions of SYSTEM.MISCINFO.

NOTE: SETUP can now be used to specify three code character sequences to be input from the keyboard (as well as two code sequences). See Section IV.1.7, "p-System Support For ANSI Terminals", for information concerning this.

NOTE: In previous versions of the UCSD p-System there were only 6 blocked devices (4, 5, 9..12). The number of blocked devices is now configurable with SETUP. After the highest numbered blocked device, subsidiary volumes are allocated device numbers. (Subsidiary volumes are described in Chapter II of this supplement.) The number of subsidiary volumes is also configurable. Above the highest numbered device set aside for subsidiary volumes, user-defined serial devices may be defined. (User-defined serial devices are also discussed in Chapter II.) The maximum number of user-defined serial devices is 16. The highest unit number allowed for any of these devices is 127. The allocation of these unit numbers is taken care of by the following fields:

FIRST SUBSIDIARY VOL NUMBER
MAX NUMBER OF SUBSIDIARY VOLS
MAX NUMBER OF USER SERIAL VOLS

These fields are described below.

NOTE: The memory update feature of Setup does not update any of the following fields:

HAS SPOOLING
HAS EXTENDED MEMORY
CODE POOL SIZE
CODE POOL BASE[FIRST WORD]
CODE POOL BASE[SECOND WORD]
SEGMENT ALIGNMENT
FIRST SUBSIDIARY VOL NUMBER
MAX NUMBER OF SUBSIDIARY VOLS
MAX NUMBER OF USER SERIAL VOLS

In order to update these fields, it is necessary to create a new SYSTEM.MISCINFO on the boot disk, and re-boot.

The following is a list of all the fields now modified by SETUP:

BACKSPACE
CODE POOL BASE[FIRST WORD]
CODE POOL BASE[SECOND WORD]
CODE POOL SIZE
EDITOR ACCEPT KEY
EDITOR ESCAPE KEY
EDITOR EXCHANGE-DELETE KEY
EDITOR EXCHANGE-INSERT KEY
ERASE LINE
ERASE SCREEN
ERASE TO END OF LINE
ERASE TO END OF SCREEN
FIRST SUBSIDIARY VOL NUMBER
HAS 8510A
HAS BYTE FLIPPED MACHINE
HAS CLOCK
HAS EXTENDED MEMORY
HAS LOWER CASE
HAS RANDOM CURSOR ADDRESSING
HAS SLOW TERMINAL
HAS SPOOLING
HAS WORD ORIENTED MACHINE
KEYBOARD INPUT MASK
KEY FOR BREAK
KEY FOR FLUSH
KEY FOR STOP
KEY TO ALPHA LOCK
KEY TO DELETE CHARACTER
KEY TO DELETE LINE
KEY TO END FILE
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
KEY TO MOVE CURSOR UP
LEAD IN FROM KEYBOARD
LEAD IN TO SCREEN
MAX NUMBER OF SUBSIDIARY VOLS
MAX NUMBER OF USER SERIAL VOLS
MOVE CURSOR HOME
MOVE CURSOR RIGHT
MOVE CURSOR UP
NON PRINTING CHARACTER
PREFIXED[DELETE CHARACTER]

```
PREFIXED[EDITOR ACCEPT KEY]
PREFIXED[EDITOR ESCAPE KEY]
PREFIXED[EDITOR EXCHANGE-DELETE KEY]
PREFIXED[EDITOR EXCHANGE-INSERT KEY]
PREFIXED[ERASE LINE]
PREFIXED[ERASE SCREEN]
PREFIXED[ERASE TO END OF LINE]
PREFIXED[ERASE TO END OF SCREEN]
PREFIXED[KEY TO DELETE CHARACTER]
PREFIXED[KEY TO DELETE LINE]
PREFIXED[KEY TO MOVE CURSOR DOWN]
PREFIXED[KEY TO MOVE CURSOR LEFT]
PREFIXED[KEY TO MOVE CURSOR RIGHT]
PREFIXED[KEY TO MOVE CURSOR UP]
PREFIXED[MOVE CURSOR HOME]
PREFIXED[MOVE CURSOR RIGHT]
PREFIXED[MOVE CURSOR UP]
PREFIXED[NON PRINTING CHARACTER]
PRINTABLE CHARACTERS
SCREEN HEIGHT
SCREEN WIDTH
SEGMENT ALIGNMENT
STUDENT
VERTICAL MOVE DELAY
```

The following are the definitions of the new fields that are affected by SETUP:

CODE POOL BASE[FIRST WORD]
CODE POOL BASE[SECOND WORD]

These two entries are used to determine where the code pool resides on machines which use extended memory.

On 8086 extended memory systems these two words, taken together, make up the 32 bit address for the base of the external code pool. The FIRST WORD is the most significant 16 bits, and the SECOND WORD is the least significant 16 bits. The least significant four bits must always be 0 on 8086 systems. Depending upon your memory configuration, you might set these values as follows for 8086 systems:

```
FIRST WORD = 1
SECOND WORD = 0
```

This indicates the binary value of 1 followed by 16 zeros (the start of the second 64K area).

On 9900 systems, the FIRST WORD is the 990/10 memory BIAS. (This is not a straight memory address; see your hardware manual for more information concerning 9900 BIAS.) It defines the start of the code pool area. The SECOND WORD is not used. There is no error checking done on this value anywhere except the 9900 hardware.

NOTE: The PoolBase field in the Pooldes record within the OS will be set to the value indicated by these two fields. If the code pool is internal (i.e., you are not using extended memory), both words must be set to 0.

NOTE: .RELPROC and .RELFUNC assembly language routines may not be executed on TI 9900 systems when an external code pool is being used. Attempting to execute such a routine will result in runtime error number 11 (instruction not implemented). Programmers should use .PROC and .FUNC which forces code to be placed in the heap (instead of the external code pool).

CODE POOL SIZE

If the code pool is external, this entry indicates the number of WORDS minus one available for it to fill. The Poolsize field in Pooldes will be set to this value. This value may be as great as 32767 (a 64K area). It may also be smaller, if desired, but it should be at least 12287 (a 24K area). The base address of this area is given by the two code pool base words. This value is ignored if you are not using extended memory.

HAS EXTENDED MEMORY

Normally the code pool resides between the Stack and the Heap. If the code pool is removed from that memory space and placed in a different area altogether, then HAS EXTENDED MEMORY should be set to TRUE, otherwise it should be set to FALSE. (An example of Extended Memory would be a 128K byte machine where the stack and heap reside within one 64K area, and the code pool within the other 64K area.)

HAS SPOOLING

If the PRINT SPOOLER (see Section 5 of this document) is to be used, this must be set to TRUE.

KEYBOARD INPUT MASK

Characters that are recieved from the Keyboard will be logically AND'ed with this value. For the typical ASCII Keyboard, this value should be set to 7F hex (which throws away the eighth bit). For some keybords which generate eight bit characters, the value FF hex should be used.

FIRST SUBSIDIARY VOL NUMBER

Subsidiary volumes are described in Chapter II of this supplement. This entry is the first unit number to be used as a subsidiary volume. If, for example, it is set to 14, the first subsidiary volume is device #14:.

NOTE: In previous versions of the UCSD p-System, only 6 blocked devices were allowed: 4, 5, 9..12. Now the number of blocked devices is configuarable. The devices from 9 through "First subsidiary vol number" - 1 are now standard blocked devices. Subsidiary volumes start with the device number indicated by "First subsidiary vol number". The number of subsidiary volumes is determined by "Max number of subsidiary vols". The highest device number allowed for subsidiary volumes, or standard blocked devices, or user-defined serial volumes (described below) is 127. (The device numbers 128 and above are reserved for user-defined devices, as described under "The Extended SBIOS" in the Installation Guide.)

WARNING: "First subsidiary vol number" must be greater than 8 to allow space for all of the standard system units.

MAX NUMBER OF SUBSIDIARY VOLS

This field indicates the maximum number of subsidiary volumes which may be online at once. Because the p-System Unit Table expands a few bytes with each additional subsidiary volume entry, this number should be set to the smallest convenient value. (Also see FIRST SUBSIDIARY VOL NUMBER.)

The highest subsidiary volume will be "First subsidiary vol number" + "Max number of subsidiary vols" - 1. This expression must be less than or equal to 127, which is the highest device number allowed for system units.

MAX NUMBER OF USER SERIAL VOLS

This entry is the total number of user-defined serial volumes desired. User-defined serial volumes are described in Chapter II of this supplement. The first device number assigned to a user-defined serial volume is "First subsidiary vol number" + "Max number of subsidiary vols" - 1.

For example, if "First subsidiary vol number" is 12 (i.e. #12:) and "Max number of subsidiary vols" is 4, then the first user-defined serial volume would be device #16:. If this entry, "Max number of user serial vols", is 2, then the user-defined serial volumes would be #16: and #17:.

If "Max number of subsidiary vols" is 0, then the first user-defined serial volume is equal to "First subsidiary vol number". In this case, "Max number of user serial vols" + "First subsidiary vol number" - 1 yields the highest numbered user-defined serial volume.

NOTE: The largest value allowed for "Max number of user serial vols" is 16. The highest numbered user-defined serial volume must be less than or equal to 127.

NOTE: User-defined serial volumes are different from user-defined devices (described under "The Extended SBIOS" in the Installation Guide). User-defined serial volumes are part of the system devices. These devices are allocated device numbers 0 through 127. Device numbers 128 through 255 are allocated for true user-defined devices. User-defined devices can only be accessed using unit I/O, whereas the standard p-System file I/O capabilities can be used with system devices such as user-defined serial volumes.

PRINTABLE CHARACTERS

This entry is used to determine which character codes will be echoed to the console. Any code, from 0 to 255 may be defined as an echoable code.

SETUP requires input in the form of a list of decimal values separated by commas or double periods. The values separated by commas correspond to the ASCII characters that will be echoed to the console. The double periods indicate that all values between the two indicated numbers are included (for example, 32..126 includes the values 32, 126, and all values between them).
The typical values will be:

13, 32..126

(13 is carriage return, 32..126 are the standard printable characters). The value 13 must always be present.

SEGMENT ALIGNMENT

This value indicates a number of bytes. The p-System is instructed to load code segments into memory starting at locations which are multiples of this number. Most systems require no special segment alignment and the values 0 or 1 indicate this. The 8086 based systems, for example, require a value of 16 here.

IV.4 Installing Turtlegraphics

IV.4.1 Introduction

Turtlegraphics has been designed to facilitate the development of portable graphics applications. Turtlegraphics is distributed in two forms. Some systems are distributed with Turtlegraphics already configured into the SYSTEM.LIBRARY and ready to run. Turtlegraphics is also sold in an adaptable form. This document describes how to install adaptable Turtlegraphics on your system.

The adaptable Turtlegraphics package is contained in the following 7 files:

GRAFIX2.CODE	{ A linkable Turtlegraphics Unit for systems using 2-word real numbers }
GRAFIX4.CODE	{ A linkable Turtlegraphics Unit for systems using 4-word real numbers }
USRGRAFS.TEXT	{ A skeleton graphics initialization unit }
USRGRAFS.CODE	{ A dummy graphics initialization unit for systems with no special setup requirements }
SYSTEM.FONT	{ A data file containing the default character font }
EXERCISE2.CODE	{ A test suite designed to exercise your graphics I/O implementation on systems using 2-word real numbers }
EXERCISE4.CODE	{ A test suite designed to exercise your graphics I/O implementation on systems using 4-word real numbers }
EXERCISE.TEXT	{ Source program for low level routine test program }

To install Turtlegraphics on your p-System it is necessary to write a collection of low-level graphics routines in assembly language and link them into one of the GRAFIX files. These routines perform simple functions such as setting a point to a specific color, or drawing a line segment. Turtlegraphics builds upon these simple routines to provide higher level services to UCSD Pascal, BASIC, and FORTRAN. If you are not already familiar with Turtlegraphics, you should stop and read its description in Chapter III of this supplement for your particular hardware. Section IV.4.2 below explains these low-level routines and the structures they manage. Section IV.4.2 also provides some implementation hints intended to help you get the best performance from your system.

Some systems require special initialization prior to performing graphics I/O. For example, it is often necessary to disable a hardware character generator on memory-mapped displays before you can write to individual screen picture elements

(pixels). Similarly, at the end of graphics I/O it is sometimes necessary to perform special operations to restore the system display to normal operation.

Such initialization and termination is handled by the initialization and termination code of the USERGRAPHICS unit. If your system requires some sort of graphics initialization or finalization you will have to develop a custom USERGRAPHICS unit. Section IV.4.3 describes how to tailor the supplied unit to suit your requirements. If your system requires no special configuration to perform graphics I/O skip Section IV.4.3 and use the dummy unit supplied.

The file *SYSTEM.FONT contains a dot matrix character representation that is used by the Turtlegraphics routines WChar and WString. Section IV.4.4 describes the structure of SYSTEM.FONT and how to build a custom version. It is strongly recommended that you do not replace the default file until the rest of Turtlegraphics is working. The EXERCISE program and Turtlegraphics error handlers expect a valid font to be available!

Section IV.4.5 describes the ways Turtlegraphics may be libraried into a p-System. It also describes the use of the EXERCISE program in debugging a Turtlegraphics adaptation.

IV.4.2 Graphics I/O Routines

The Turtlegraphics unit is created by linking seven assembly code I/O routines into either GRAFIX2.CODE or GRAFIX4.CODE. These routines interact not only with the system display, but also with a collection of data structures that describe the state of Turtlegraphics.

None of the routines described in this section need to perform range-checking on the parameters passed, EXCEPT for Draw_Line. When any of these routines (except Draw_Line) are to be called, Turtlegraphics performs the appropriate range-checking beforehand.

The following subsections describe the syntax and semantics of these routines:

Procedure Query_Environment (var DisplayDesc: DisplayRec);
Turtlegraphics uses this procedure to initialize the parameters that describe the target configuration. Query_Environment is passed a pointer to a record that describes the machine-dependent aspects of the system. ALL the fields must be filled by this routine. The Pascal description of the record below comes from Turtlegraphics.

```
DisplayRec =
  record
    XPixelCnt: integer; {number of pixels in the x direction
                         on the actual display}
    YPixelCnt: integer; {number of pixels in the y direction
                         on the actual display}
    MaxColor: integer; {maximum valid color number}
    AspectX: integer;
    AspectY: integer; {a pair of integers such that
                       the ratio: AspectX/AspectY, is the
                       aspect ratio of the actual physical
                       display}
    CharHeight: integer;
    CharWidth: integer; {specifies the height and width of
                         characters generated by SYSTEM.FONT
                         in pixels. For the SYSTEM.FONT
                         shipped, the default is 8x8}
    TargetStamp: integer; {identifies the current target
                           machine configuration. Used as a
                           validity check by LOADFIGURE,
                           GETFIGURE, and PUTFIGURE }
  end;
```

Function Figure_Size (Screen: ScreenPtr): Integer;
This function tells Turtlegraphics the number of words required to store the figure described by the indicated ScreenRec on the target machine. This function is called by CREATE_FIGURE when an application dynamically creates a figure. The size of the figure varies. Typically it is a function of the figure area times the number of colors available.

The encoding of user-created figures is completely managed by the low-level routines you are writing. You may elect to encode your figures for maximum data compression if your applications store many figures. You may encode for maximum update efficiency, if you have a great deal of available storage.

On systems with large physical memory capacity, you may elect to store the first several user figures outside of the Stack/Heap address space. In that situation, the figure size can be zero.

The type ScreenPtr is a pointer to a Pascal record that describes the state of a Turtlegraphics figure. There is one screen description record for every Turtlegraphics figure, including the actual display.

```
ScreenPtr = ^ScreenRec;

ScreenRec =
    record
     Valid: ScreenPtr;  {pointer should always be a self
                              reference when figure is valid}
     FigPtr: ^fig; {pointer to the figure's locn in memory;
                    a nil pointer indicates that the record
                    describes the actual display}
     Color: integer; {current pen color}
     Backgnd: integer; {current turtle background color}
     Mode: integer; {current turtle drawing mode}
     {the next four values delimit the viewport by pixel values:}
     MinXPix: integer;
     MinYPix: integer;
     MaxXPix: integer;
     MaxYPix: integer;
     XPix: integer; {turtle pixel x position}
     YPix: integer; {turtle pixel y position}
     TargetStamp :integer;
               {target machine stamp which identifies the
               machine configuration upon which the figure
               was created; it is updated only by low-level
               routines}
     Size: integer; {size of the figure in words}
```

```
    XPos: real; {turtle x posn in display scale units}
    YPos: real; {turtle y posn in display scale units}
    Heading: real; {current orientation of the turtle}
    ScaleStamp: integer;
                {Specifies the scale generation value
                for which XPos and YPos are valid}
end;
```

Function Read_Screen_Pixel
(Pointer: ScreenPtr;
XPixel, YPixel: Integer): Integer;

The Read_Screen_Pixel function returns the color of the pixel at the specified location in figure. The XPixel and YPixel parameters give the pixel location. Turtlegraphics checks the range on all calls to this routine. If the FigPtr in the indicated screen record is nil, then the function should return the state of the actual display.

Procedure Set_Screen_Pixel (Pointer: ScreenPtr;
XPixel, YPixel: Integer;
Shade: Integer);

The Set_Screen_Pixel procedure sets the pixel at (XPixel, YPixel) to the designated color. Shade specifies the color value. If the FigPtr in the indicated screen record is nil, then the procedure should modify the actual display.

Procedure Comp_Screen_Pixel (Pointer: ScreenPtr;
XPixel, YPixel: Integer);

The Comp_Screen_Pixel procedure complements the pixel at (XPixel, YPixel). Shade specifies the color value. If the FigPtr in the indicated screen record is nil, then the procedure should modify the actual display.

The definition of complement is left to the discretion of the implementor for a given target machine, given the following constraints: Complementing a pixel must result in a different unique color, the complement of which is the original color. This implies that a machine which supports Turtlegraphics must have an even number of colors in its palette, or that only an even number can be used.

Procedure Fill_Color (Screen: ScreenPtr; Shade: Integer);

This procedure fills a portion of the specified screen with the designated color value. The contents of the screen record fields MaxXPix, MinXPix, MaxYPix, and MinYPix describe the rectangular area that is filled. The FigPtr contains a pointer to the area of memory in which the figure is stored. If FigPtr is nil, the actual display (or a portion of it) should be filled.

Procedure Draw_Line (Pointer: ScreenPtr;
StartX, StartY, EndX, EndY: Integer);

Draw_Line is the most complex routine to be written for Turtlegraphics. It must draw a line segment in the specified screen. The starting and ending points of the line segment are described by (StartX, StartY), (EndX, EndY). **Important:** Turtlegraphics does no range checking on the line segment! The implementor is responsible for computing all the points in the line segment, and ONLY plotting those within the viewport. (The viewport is defined by the screen record fields MaxXPixel, MaxYPixel, MinXPixel and MinYPixel). In addition, the points on the line must be plotted using the PenColor and Mode specified in the screen record. The valid mode values and their meanings are described below:

```
const
    nop = 0;
    substitute = 1;
    overwrite = 2;
    underwrite = 3;
    complement = 4;
```

If the current mode is nop, drawline is NOT called.

Substitute mode calls for every visible point on the line segment to be unconditionally plotted.

Overwrite, for the purposes of drawline, is the same as Substitute.

Underwrite mode indicates that visible points on the line segment are plotted only if the pixel at that location is currently set to the Backgnd color, as described in the screen record.

Complement mode indicates that the visible points on the line segment should be complemented using the definition of complement used by the Comp_Screen_Pixel procedure.

The performance of Turtlegraphics is strongly influenced by the efficiency of these routines. It is recommended that every effort be made to optimize their operation. Computing the line trajectory and then only performing simple addition to determine the locus of the next point is a good way to minimize computing. A "psuedo-Pascal" procedure below outlines how Draw_Line might be structured:

```
procedure Draw_Line
                (Screen: ScreenPtr;
                StartX, StartY, EndX, EndY: integer);

var Temp, X, Y, DeltaX, DeltaY, Cnt,
    XInc, XResidue, XCorrection,
    YInc, YResidue, YCorrection: integer;

    DXNeg, DYNeg: Boolean;

procedure exchange_xy;
  begin
    Temp := StartX;
    StartX := EndX;
    EndX := Temp;
    DeltaX := -DeltaX;
    Temp := StartY;
    StartY := EndY;
    EndY := Temp;
    DeltaY := -DeltaY;
    DXNeg := not DXNeg;
    DYNeg := not DYNeg;
  end;

procedure update_pix (px, py: integer);
  begin
    with Screen^
    do begin
      if (px>=MinXPix) and (px<=MaxXPix) and
        (py>=MinYPix) and (py<=MaxYPix)
      then
        case mode of
          substitute, overwrite:
            set_screen_pixel(Screen,px,py,color);
          underwrite:
            if read_screen_pixel(Screen,px,py)=backgnd
            then set_screen_pixel(Screen,px,py,color);
          complement:
            comp_screen_pixel(Screen,px,py);
          end;
      end;
  end;

begin
  DeltaX := EndX - StartX;
```

```
DeltaY := EndY - StartY;
DXNeg := DeltaX<0;
DYNeg := DeltaY<0;
if DeltaX = 0 then {vertical line}
    begin
      if DYNeg then exchange_xy;
      for Y:=StartY to EndY
      do update_pix (StartX, Y)
    end
else if DeltaY = 0 then {horizontal line}
    begin
      if DXNeg then exchange_xy;
      for X:=StartX to EndX
      do update_pix (X, StartY)
    end
else if abs(DeltaY) > abs(DeltaX) then
{abs(slope) > 45 degrees}
    begin
      if DYNeg then exchange_xy;
      {substitute for fractions:}
      YInc := abs((64 * DeltaY) div DeltaX);
      {using binary fixed point arithmetic operations:}
      YCorrection := YInc mod 64 + 1;
      YInc := YInc div 64;
      Y := StartY;
      X := StartX;
      YResidue := 0;
      Cnt := 0;
      while Y <= EndY;
      do begin
          update_pix (X, Y);
          Y := Y+1;
          Cnt := Cnt+1;
          if Cnt=YInc then
            if YResidue > 64 then
              begin
                Cnt := Cnt -1;
                YResidue := YResidue - 64
              end
            else
              begin
                YResidue := YResidue + YCorrection;
                Cnt := 0;
                if DXNeg then X := X-1
                  else X := X + 1;
              end;
```

```
            end;
          end
      else {abs(slope)<= 45 degrees}
          begin
            if DXNeg then exchange;
            {substitute for fractions:}
            XInc:= abs((64 * DeltaX) div DeltaY);
            {using binary fixed point arithmetic operations:}
            XCorrection := XInc mod 64 + 1;
            XInc := XInc div 64;
            X := StartX;
            Y := StartY;
            XResidue := 0;
            Cnt := 0;
            while X <= EndX
            do begin
                update_pix (Y, X);
                X := X+1;
                Cnt := Cnt+1;
                if Cnt=XInc then
                  if XResidue > 64 then
                    begin
                      Cnt := Cnt -1;
                      XResidue := XResidue - 64
                    end
                  else
                    begin
                      XResidue := XResidue + XCorrection;
                      Cnt := 0;
                      if DYNeg then Y := Y-1
                        else Y := Y + 1;
                    end;
                end;
          end;
    end;
```

IV.4.3 Graphics System Initialization

It is frequently necessary to perform some special operations to ready a system to display graphic information. On some systems, for example, the display hardware must be switched into a different mode. Similarly, at the termination of graphic I/O, it is often necessary to perform some operations to restore the system to normal operation.

Turtlegraphics addresses this situation by expecting a unit called USERGRAPHICS to be in *SYSTEM.LIBRARY. This unit has one procedure,

```
procedure Hardware_config;
```

When Turtlegraphics is performing initialization it calls Hardware_config. At the end of a program, any termination code present in the USERGRAPHICS unit is executed.

Turtlegraphics is shipped with a skeleton version of USERGRAPHICS in the file USRGRAFS. This may be used if no special initialization or termination are required. If your system requires special configuration, you can write your own USERGRAPHICS unit. The only requirement is that USERGRAPHICS be in *SYSTEM.LIBRARY, and that the FIRST procedure in its interface section must be called Hardware_config.

IV.4.4 Turtlegraphics Character Fonts

Turtlegraphics allows programs to label figures by calling two special routines, WChar and WString. These routines draw characters in figures by using a table stored in a file called *SYSTEM.FONT.

The standard system is shipped with a character font that contains 128 ASCII codes, similar in style to those on the some personal computers. Each character occupies an area 8 pixels high by 8 pixels wide. This character size may be inappropriate to some displays. On high resolution displays, such characters are too small. On low resolution displays, it may be desirable to use a 5x7 character matrix.

To replace the default font with one of your own design, you must first be sure that your version of the function Query_Environment initializes the display record fields CharHeight and CharWidth to the proper values. You must then generate a new table and save it on the boot disk as SYSTEM.FONT.

A Font Structure

Turtlegraphics reads the font table as a 1-dimensional packed Boolean array. To draw a character, it computes the index of the first bit of a character as follows:

```
index:= ord(character) * CharHeight * CharWidth;
```

It then displays the characters, using an algorithm similar to:

```
for x:=0 to CharWidth - 1
do for y:=0 to CharHeight - 1
    do if font[index + x*CharHeight + y] then
            set_pixel(screen, x+turtle_x, y+turtle_y, 1)
        else
            set_pixel(screen, x+turtle_x, y+turtle_y, 0);
```

Therefore, the font table is designed like a

```
packed array [0..127, 0..CharWidth-1, 0..CharHeight] of Boolean
```

Do not use such a declaration to create your character font in Pascal! Pascal aligns all arrays (packed arrays included) so that all rows and columns begin on word boundaries. This will cause you problems if the product of CharHeight and CharWidth is not evenly divisible by 16!

IV.4.5 Linking and Librarying Turtlegraphics

Once you have written the low-level graphics I/O routines, you must link them into one of the GRAFIXx.CODE files to produce a complete Turtlegraphics unit. The standard p-System Linker will do the job quite nicely. Select either GRAFIX2.CODE or GRAPHIX4.CODE to host your linking, depending on the real number size of your P-machine. The output file should be called TURTLE.CODE.

To run programs that USE TURTLEGRAPHICS, be sure SYSTEM.FONT is on the boot disk. Also, be sure that *USERLIB.TEXT indicates where TURTLEGRAPHICS and USERGRAPHICS may be found, or include both units in *SYSTEM.LIBRARY.

Exercising TURTLEGRAPHICS

Included in the adaptable Turtlegraphics package are two exercise programs, EXERCISE2.CODE and EXERCISE4.CODE. Both are created from the source in EXERCISE.TEXT. They are provided to help you debug your low-level graphics I/O routines. They are programs written in Pascal, and are designed to exercise progressively more sophisticated aspects of your routines.

The exercise is divided into two main sections. The first section tests graphics I/O to the actual display. The second half runs a similar set of test on user-created figures, and then copies the figures to the actual display for your examination. The paragraphs that follow explain the operation of the exercises.

Actual Display Set and Clear Pixel Test

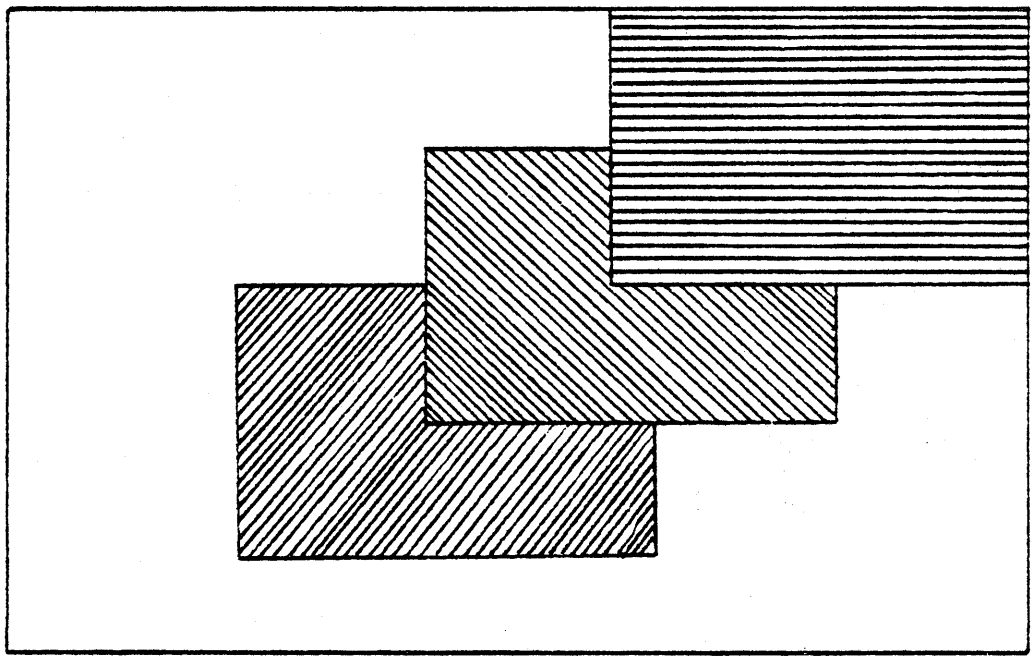
This test should display a set of colored, dotted lines horizontally across the display, drawing a pair from left to right with each pass. The test should end when it has cycled through all the colors available on your display. It will then query you to see that it has determined the number of available colors correctly. For this and all subsequent queries, affirm a correct result by pressing 'Y' (either upper or lower case). Any other character is considered a negative response, and the exercises will terminate.

Actual Display Fill_Color Tests

The next set of exercises tests Fill_Color. First the system calls Fillscreen in all the valid colors for your system. You should be careful to be sure that Set_screen_pixel uses the same color values as Fill_Color.

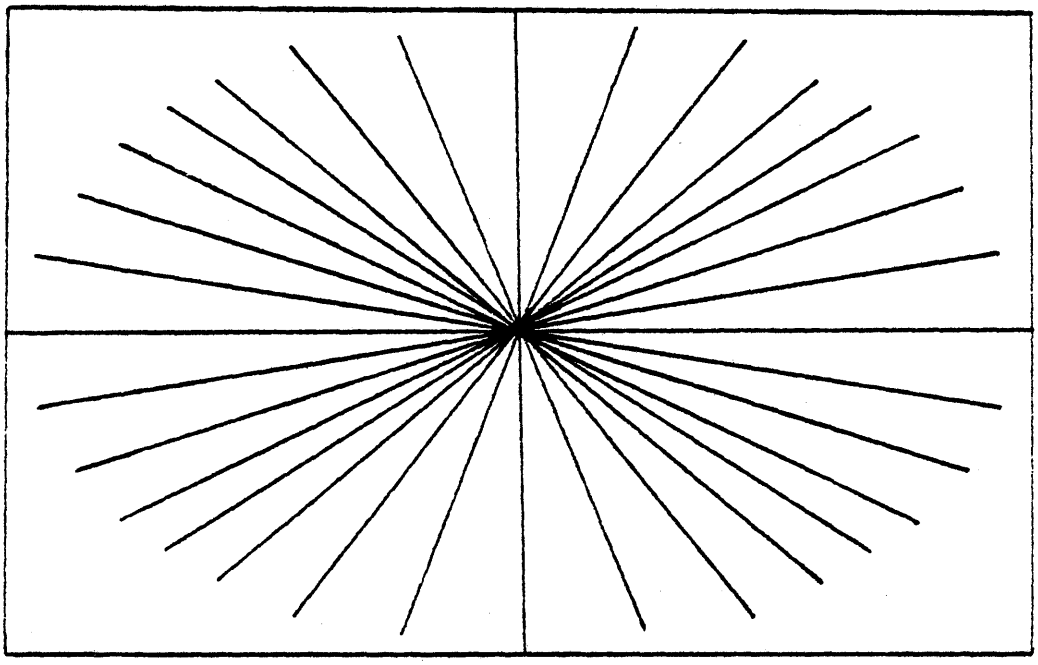
The next phase of this test should set the screen to color 0 and then display a set of overlapping rectangles from the lower left hand corner of the display to the upper right, using all the available colors. This is a test of windowing in Fill_Color.

The screen for a 4-color system is shown below:

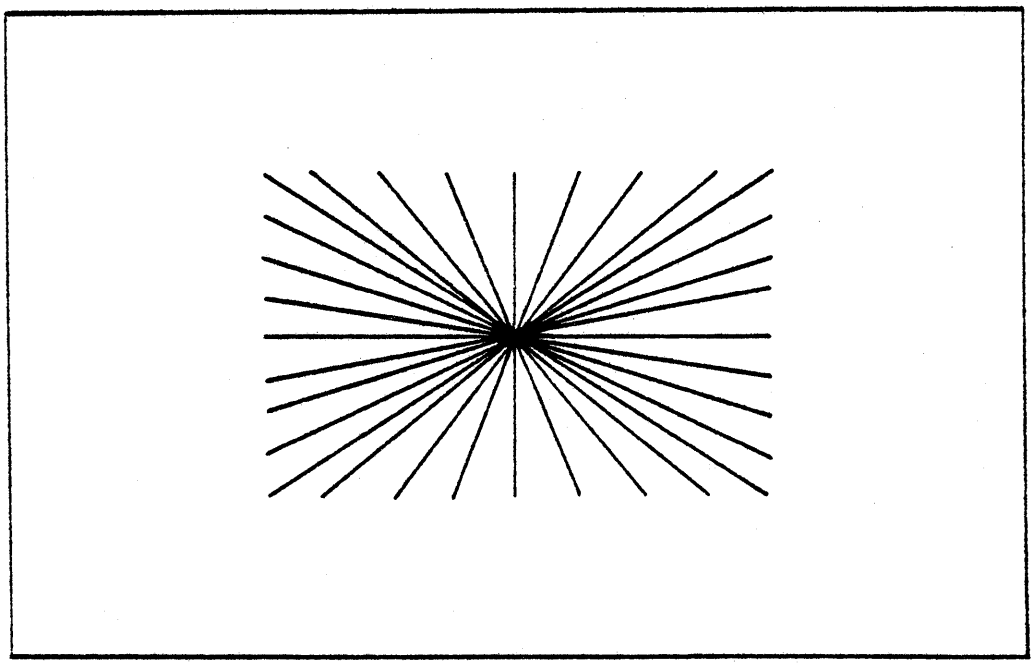


Actual Display Line-Drawing Exercises

The next portion of the exercises are designed to check the Draw_line routine. First a set of radial lines are drawn from the center of the screen. Thirty-six (36) radials are drawn starting at the 3 O'clock position, then moving counterclockwise around the center point. This behavior is repeated for all the non-zero colors. Again, be sure that the color assignment matches both Fill_Color and Set_screen_pixel. A sample copy of the sort of display created by this test appears in the figure below:



The next part of the tests check to see if Draw_line respects the viewport of the display. The system issues the same commands as it did on the previous tests, but this time the viewport is restricted to a small rectangle in the center of the display. The result should be that the lines should stop at the periphery of the rectangle, rather than continuing to their previous end points. The figure below show how the display should appear when the test in complete:



The last line-drawing test on the actual display is performed only on systems with more than two colors. This checks the update modes for line drawing. A small rectangular area in the middle of the display is shaded. Then the same set of radials as before are drawn in each of the modes. The expected effects for each mode are summarized here:

Mode 0
In Nop mode nothing else should appear.

Modes 1 and 2
Both Substitute and Overwrite modes should draw lines over the top of the rectangle and beyond.

Mode 3
Underwrite mode should not alter the center rectangle. Radials should be visible from the periphery of the rectangle, continuing out to their previous end points.

Mode 4
In complement mode the radials should emerge from the center point, but change color at the periphery of the center rectangle, and terminate at the edge of the screen.

User-Created Figures Exercises

After testing the display, the Exercise program performs all the same operations on user-created figures. The results of each test are indicated on the actual display. A frame is constructed on the actual display using Fill_Color. The user figures resulting from each test are copied into this viewing frame.

All the same figures should result, except for those that tested viewports: a user-created figure cannot contain a viewport.

We remind you that a test can prove the presence of bugs, but never their absence! EXERCISE will not prove that your routines are error free, but if all the tests execute successfully, your low-level routines work well enough that you can now start using Turtlegraphics.