SAGE COMPUTER

p-System Program Development

Copyright 1983 by SofTech Microsystems, Inc.

All rights reserved. No part of this work may be reproduced in any form or by any means or used to make a derivative work (such as a translation, transformation, or adaptation) without the written permission of SofTech Microsystems, Inc.

p-System is a trademark of SofTech Microsystems, Inc.

UCSD and UCSD Pascal are registered trademarks of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use is contrary to the laws of the State of California.

Printed in the United States of America.

Disclaimer

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

PREFACE

This book is a reference manual for the p-System . It describes the p-System facilities which enable you to develop programs. It is designed for a data processing professional who is familiar with the p-System. The details of the various programming languages—UCSD Pascal , FORTRAN, and BASIC—aren't covered in this manual.

The following books and manuals may also be of interest to you. They are available from SofTech Microsystems.

Personal Computing with the UCSD p-System
UCSD Pascal Handbook
Operating System Reference Manual
Assembler Reference Manual
Optional Products Reference Manual
Internal Architecture Reference Manual
Adaptable System Installation Manual
FORTRAN-77 Reference Manual
BASIC Reference Manual

T A B L E

O F

C O N T E N T S

Table of Contents

Table of Contents

Table of Contents

Table of Contents

CHAPTER 1

INTRODUCTION

HOW TO USE THIS MANUAL

This book is a reference manual for use with the p-System. It describes the p-System facilities which enable you to develop programs.

Chapter 2, "Compiling Programs," covers the Pascal compiler. The UCSD Pascal programming language isn't covered in this manual. You should see the UCSD Pascal Handbook if you are interested in a thorough description of the language. This chapter also describes units, segments, and libraries. These facilities are used when you separately compile program modules. Using them you can compile and run much larger programs than you would otherwise be able to within a given computer's memory and disk space limitations.

Chapter 3, "User Interfacing," describes several p-System facilities that can assist your programs in presenting a clean and portable user interface. For example, the p-System can be completely hidden underneath your application's own environment. Programs may be chained together and called from a simple menu driver that appears when a disk is bootstrapped. Whether or not you use this approach, you may wish to take advantage of screen handling and error interception facilities described in this chapter.

Chapter 4, "File Management," covers the file management units. These allow your programs to manipulate disk files in a similar fashion to the filer. For example, files can be listed and removed, volumes can be crunched, and so forth.

Chapter 5, "Debugging and Analysis" covers the debugger and the performance monitor units. The debugger is a very powerful tool for finding and correcting errors that might exist in programs you write. The performance monitor allows you to accumulate statistical information concerning various performance-related issues. Many of the utilities described in Chapter 6 are also valuable as debugging and analysis aids.

Chapter 6, "Utility Programs," describes several p-System utility programs that are generally useful. Most of these are specifically valuable during program development.

BACKGROUND

In June 1979, SofTech Microsystems in San Diego, began to license, support, maintain, and develop the p-System. The resulting effort to build the world's best small computer environment for executing and developing applications has dramatically increased the growth and use of the p-System. The first p-System ran on a 16-bit microprocessor. Today, the p-System runs on 8-bit, 16-bit, and 32-bit machines—including the Z80, 8080/8085, 8086, 6502, 6809, 68000, 9900, PDP-11, LSI-11, and VAX.

The p-System began as the solution to a problem. The University of California at San Diego needed interactive access to a high-level language for a computer science course. In late 1974, Kenneth L. Bowles began directing the development of the solution to that problem: the p-System. He played a principal role in the early development of the software.

In the summer of 1977, a few off-campus users began running a version of the p-System on a PDP-11. When a version for the 8080 and the Z80 began operating in early 1978, outside interest increased until a description of the p-System in Byte Magazine drew over a thousand inquiries.

As interest grew, the demand for the p-System couldn't be met within the available resources of the project. SofTech Microsystems was chosen to support and develop the p-System because of its reputation for quality, high technology, and language design and implementations.

DESIGN PHILOSOPHY

The development team, many of whom continued their efforts on behalf of the system at SofTech Microsystems, decided to use stand-alone, personal computers as the hardware foundation for the p-System rather than large, time-sharing computers. They chose Pascal for the programming language because it could serve in two capacities: the language for the course and the system software implementation language.

The development team had three primary design concerns:

1. The user interface must be oriented specifically to the novice, but must be acceptable to the expert.

2. The implementation must fit into personal, stand-alone machines (64K bytes of memory, standard floppy disks, and a CRT terminal).

3. The implementation must provide a portable software environment where code files (including the operating system) could be moved intact to a new microcomputer. In this way, application programs written for one microcomputer could run on another microcomputer without recompilation.

The current design philosophy at SofTech Microsystems, where the p-System continues to evolve, is bascially the same as the original philosophy.

User-Friendly

The p-System continuously identifies its current mode and the options available to you in that mode. This is accomplished by using menus, displays, and prompts. You may select an option from a menu by pressing a single-character activity. The system's displays then guide your interactions with the computer. As you gain more experience, you can ignore the continuous status information—unless it is needed.

Portability

The p-System is more portable than any other microcomputer system. It protects your software investments without restricting hardware options. The p-System does this by compiling programs into p-code—rather than native machine language—thus, allowing these code files to be executed on any microcomputer that runs the p-System.

CHAPTER 2

COMPILING PROGRAMS AND UNITS

INTRODUCTION

This chapter is principally concerned with the UCSD Pascal compiler. The BASIC and FORTRAN compilers are described in separate reference manuals. However, this chapter should be useful even if you are only planning to use BASIC or FORTRAN.

The UCSD Pascal programming language isn't covered here. If you're interested in a detailed description of UCSD Pascal, you should consult the UCSD Pascal Handbook.

Separate compilation is also covered in this chapter. Specifically, the UCSD Pascal unit construct, program segmentation, and code file libraries are addressed.

USING THE COMPILER

The compiler takes a text file as input and generates a machine-portable code file as output. The generated code file contains p-code, which is executed by the p-System's p-machine emulator. This emulator is written in assembly language and runs directly on the host computer's hardware.

You can start the compiler by selecting the C(ompile or R(un activity of the Command menu. If a work file exists, it is compiled. Otherwise, you are prompted for a text file to compile, like this:

```
Compile what text ? _
```

Enter the name of the text file, but don't include the ".TEXT" suffix (which is assumed). Next, you are asked:

```
To what codefile ? _
```

Here, you should enter the name of the code file that you want the compiler to produce. Don't include the ".CODE" suffix (which, once again, is assumed). If you enter '$' followed by <return>, the code file is given the name which corresponds to the text file being compiled. If you simply press <return>, the code file *SYSTEM.WRK.CODE is produced. The next prompt is:

```
Output file for compiled listing ? (<esc> for none) _
```

This allows you to indicate where you want the compiled listing to be sent. You can respond with a file name, with a communications volume such as PRINTER: or CONSOLE:, or simply with <return>. When you enter a file name, the listing is placed in the file. You may use the suffix ".TEXT", but it is always appended if you don't. If you specify a communications volume, the listing is sent there (where it is printed, displayed, or transmitted). When you simply press <return>, no listing is produced.

The $L Pascal compiler option can also create a compiled listing, as described later in this chapter. If you indicate a file or communications volume in response to this prompt, however, the compiler option is overriden. (You should note that ".TEXT" isn't automatically appended with the $L optior as it is with this prompt.)

While the compiler is running, it displays a report of its progress on the screen in this manner:

```
Pascal compiler - release level VERSION
<   0>.....................
INITIALIZE
<  19>.....................................
AROUTINE
<  61>.............................................
< 111>.......
MYPROG
< 119>.........................................

  237 lines compiled

INITIALI .
MYPROG   ..
```

During the first pass, the compiler displays the name of each routine. In this example, INITIALI, AROUTINE, and MYPROG are the routines. The numbers enclosed within angle brackets, < >, are the current line numbers, and each dot on the screen represents one source line compiled.

During the second pass, the names displayed are segments. In the example, MYPROG is the program segment, and INITIALI is a segment routine. Here the dots represent one routine within the segment. MYPROG contains both itself and AROUTINE.

You can suppress this output if you want by using the $Q compiler option, described later.

If the compilation is successful, that is, if no compilation errors are detected, the compiler creates a code file. This file is called *SYSTEM.WRK.CODE if you are using work files or if you press <return> in response to the compiler's prompt:

```
To what codefile?
```

Otherwise, it is given the name that you specify in response to that prompt.

When you select R(un (instead of C(ompile), the resulting code file is automatically executed. If you have a work code file, or if you have just compiled a program, R(un simply executes it.

Syntax Errors

If your program text doesn't conform to the rules of the Pascal programming language, the compiler issues a syntax error. When this happens, the text where the error occurred is displayed, along with an error number or message. Here are two examples:

```
MY FIRST LINE OF TEXT <---
'PROGRAM' or 'UNIT' expected
Line 1
Type <sp> to continue, <esc> to terminate, or 'e' to edit


MY FIRST LINE OF TEXT <---
Error #405
Line 1
Type <sp> to continue, <esc> to terminate, or 'e' to edit
```

This is the same error displayed twice (the first line of a program is incorrect). In the first case, the error message is displayed. In the second case, the error number is displayed. You only receive the error message if the file *SYSTEM.SYNTAX is available. If *SYSTEM.SYNTAX isn't present, you need to look up the error number in the appropriate appendix to this manual. (Compiler error messages are given for Pascal, BASIC, and FORTRAN.)

After each syntax error, a message like one of these is displayed and the compiler gives you the option of pressing <space> to continue the compilation, <esc> to terminate it, or 'E' to enter the editor.

You can press <space> for every syntax error in the program if you wish. In this way, you can usually discover all of the errors that exist. (However, some syntax errors can "confuse" the compiler and hide other syntax errors.) A code file is never produced if syntax errors are found. But, a compiled listing can be produced. You can use such a listing to keep track of the errors so that you can correct them all at once.

If you elect to press 'E' after a syntax error, the compilation is terminated (as it is with <esc>). However, you can now fix the error immediately because the editor is automatically invoked. If the file that you are compiling is a work file, it is read into your work space. If it isn't a work file, you are asked to specify which file you want to edit (in the editor's normal fashion). In either case, when the file is read into the work space, the cursor is placed at the exact spot where the error was detected. The error message or number is redisplayed and you must press <space> to begin editing so that you can fix the problem. When a syntax error occurs in an include file (see the $I compiler option), you must be sure to specify that file correctly as you enter the editor. You are informed of the name of the include file after the "Line #" portion of the syntax error message.

If both the $Q and $L compiler options are in effect, the compilation continues and the syntax error is only reported in the listing file. In this case, the screen remains undisturbed by syntax errors.

Compiled Listings

The compiler may optionally produce a listing of the compiled source. This listing contains your text along with information about the compilation. Compiled listings are very useful for reference as well as analysis and debugging purposes.

In order to produce a compiled listing, you can use the compiler's prompt for a listing file which is described above. Alternatively, you can use the $L compiler option which is described under compiler options, below.

Here is the entire compiled listing for a very simple program:

```
Pascal Compiler VERSION        1/01/83          Page  1


    1   0    0:d    1  {$L list.text}
    2   2    1:d    1  Program Comp_Listing_Example;
    3   2    1:0    0  Begin
    4   2    1:0    0  {
    5   2    1:0    0   This is an example listing of
    6   2    1:0    0   an empty program.
    7   2    1:0    0  }
    8   2     :0    0  End.

End of Compilation.
```

Here is a sample portion of a more complex listing:

```
393  10   12:d     1 Procedure iocheck;
{ commented out ';' }{;
{ commented out ';' } This procedure will check the i/o operations of the
{ commented out ';' } index as it is in the process of rebuilding
397  10   12:d     1 }
398  10   12:0     0 Begin
399  10   12:1     0   If ioresult <> 0 Then
400  10   12:2     6     Begin
401  10   12:3     6       p1              := 'index I/O failure.';
402  10   12:3    32       prompt(errorline);
403  10   12:2    38     End; { if ioresult <> 0 then }
404  10   12:0    38 End; { iocheck }
405  10   12:0    50
406  10   13:d     1 Procedure dropindex(position: isamcoverage);
```

In those lines that aren't marked as commented out (which is intended to warn you that a comment may have accidentally eliminated some Pascal code), the numbers that precede a source line are:

1. The line number. For example, 397 in the listing above.

2. The Pascal segment number. This entire example is part of segment number 10.

3. The routine number followed by a colon and the "lex level." In the example, procedure iocheck is routine number 12 and procedure dropindex is routine 13. The lex level indicates how deeply the text is nested within Pascal begin-end pairs.

4. The number of bytes of data or code storage which the routine requires at that point. For example, the IF statement, line 399, requires 6 bytes of p-code. The entire procedure iocheck requires 50 bytes of p-code.

Lines which contain declarations (variables, constants, and so forth) show a "d" following the routine number. In the listing above, lines 393 and 397 are examples of this.

When the module that you are compiling uses a unit, the interface section of that unit appears in the compiled listing with a "u" where the "d" normally appears. Also, the additional line 'USING <UNITNAME>' appears in the heading to make it easier for you to distinguish interface sctions from the text that you are specifically compiling.

Here is a portion of a compiled listing which shows syntax errors:

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

This shows two instances of error 104. This particular error indicates that an undeclared identifier was found—"lastpageitem" is the problem in both cases. An actual message indicating "undeclared identifier" would have been listed if the file *SYSTEM.SYNTAX had been available.

Error messages indicate the position of the previous syntax error. In this example, line 596 contains the first syntax error and line 602, which contains the second, references line 596 as the previous syntax error.

Compiler Options

You may direct some of the compiler's actions by the use of compiler options embedded in the source code. Compiler options are a set of commands that may appear within "pseudo-comments." A pseudo-comment is like any other Pascal comment (it is surrounded by '(*' and '*)', or by '{' and '}'). However, a dollar sign immediately follows the left-hand delimiter, for example:

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

There are two kinds of compiler options: "switch" options and "string" options. A switch option is a letter followed by a '+', '-', or '^'. A string option is a letter followed by a string. (In the examples above, the second is a string option; the others are switch options.) A pseudo-comment may contain any number of switch options (separated by commas), and zero or one string options. If a string option is present in a pseudo-comment, it must be the last option. The string is delimited by the option letter and the end of the comment.

If the pseudo-comment uses '(*' and '*)', the string in a string option may not contain an '*'.

Some options may appear anywhere within the source text. Others must appear at the beginning of the file (before the reserved word PROGRAM or UNIT).

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

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

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

The individual compiler options are described below in alphabetical order. If you don't use any compiler options, their default values will be in effect. Here are the default values for the compiler options:

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

These remain in effect unless you override them.

The $Q option defaults to Q- if HAS SLOW TERMINAL is false and Q+ if HAS SLOW TERMINAL is true. (HAS SLOW TERMINAL is a data item in SYSTEM.MISCINFO which indicates whether or not you have a hard copy terminal or a screen).

Conditional compilation is also controlled by compile-time options as described below.

$B - Begin Conditional Compilation

$B is a string option. It starts compilations of a section of conditionally compiled source code. See the section on conditional compilation, below.

$C - Copyright Field

$C is a string option. It places the string directly into the copyright field of the code file's segment dictionary. The purpose of this is to have a copyright notice embedded in the code file.

$D - Conditional Compilation Flag

$D is a string option. It is used to declare or alter the value of a conditional compilation flag. See the section on conditional compilation, below.

$D - Symbolic Debugging

There are two $D compiler options. This one is a switch option. $D+ turns on symbolic debugging information. $D- turns off symbolic debugging information. The default is $D-. (See Chapter 5 for more informaton about this compiler option.)

$E - End Conditional Compilation

"E" is a string option. It ends a section of conditionally compiled source code.

$I - I/O Check Option

There are two options named by $I. The first is a stack switch option (IOCHECK).

$I+, which is the default, instructs the compiler to generate code after each I/O statement in a program. This code verifies, at run-time, that the I/O operation was successful. If the operation wasn't, the program terminates with a run-time error.

$I- instructs the compiler not to generate any I/O checking code. In the case of an unsuccessful I/O operation, the program continues.

When you use the $I- option, your programs should specifically test IORESULT (an intrinsic p-System function) when there is the chance of an I/O failure. If $I- is used and you don't test IORESULT, the effects of an I/O error are unpredictable.

During program development you should probably use $I+. When your program is thoroughly debugged, you may wish to use $I- since less memory space is required without the I/O checking code. Also, you may wish to intercept I/O errors in your program. (For example, you may enter something incorrect from the keyboard. Rather than terminating with an I/O error, your program could prompt you to correct the problem and try again.)

$I - Include File

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

```
{$I PROG2}
```

This includes the file PROG2.TEXT in the program's source.

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

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

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

Note that if a file name begins with a '+' or '-', a blank must be inserted between the letter 'I' and the string. For example:

```
(*$I +PROG2*)
```

$L - Compiled Listing

You may use $L either as a toggle switch option or a string option. When used as a toggle, it turns the listing on or off at that point in the source text. When used as a string option, it indicates the name of the listing file.

Here are two examples of $L with a string option:

```
(*$L LIST.TEXT*)
(*$L PRINTER:*)
```

The first example indicates that the compiled listing is to be saved on disk as the file LIST.TEXT. The second example sends the listing to the printer.

When used as a toggle, $L+ turns the listing on and $L-turns it off. Using these options, you can list only parts of a compilation if you wish. The default for the toggle is $L- if you have not named a listing file using the compiler's prompt or using $L with a string option. The default is $L+ if you have named a listing file in either of these ways. No matter which way you name the listing file, you can switch the listing on or off using $L+ or $L-.

If you don't specifically name a listing file and $L+ is in effect, the compiler writes to *SYSTEM.LST.TEXT.

You should note that listing files which are sent to disk files may be edited as any other text file, provided they are created with a .TEXT suffix. Without the .TEXT suffix, the p-System treats the listing as a data file. With the $L option, .TEXT is never appended. However, from the compiler's prompt for a listing file, .TEXT is always appended (unless you enter it specifically).

$N - Native Code Generation

This is a swtich option. $N+ outputs compiler information which allows code generation to take place. $N- doesn't output this information. The default is $N-. This option is discussed in the Native Code Generator section which is part of the Optional Products Reference Manual.

$P - Page and Pagination

The compiler can place page breaks in the compiled listing. It does this so that listings sent to the printer (or listings sent to files and later T(ransferred to the printer) break across page boundaries. A form feed character (ASCII FF) is output every 66 lines if $P+ is in effect (this is the default). If you don't want this, you

should use $P-.

You can specifically cause a page break at any point in a compiled listing by using the $P option without a plus or minus sign.

$Q - Quiet

This is used to suppress the compiler's standard output to the console. $Q+ causes the compiler to suppress this output and $Q- causes it to resume outputting status information.

$R - Range Checking

$R is a stack switch option. The default, $R+, causes the compiler to output code after every indexed access (for example, to Pascal arrays) to check that it is within the correct range. This is called range checking. $R- turns range checking off.

Programs compiled with the $R- are slightly smaller and faster since they require less code. However, if an invalid index occurs or a invalid assignment is made, the program isn't terminated with a run-time error. Until a program has been completely tested, it is suggested that you compile with the R+ option left on.

$R2 and $R4 - Real Size

$R2 causes the code file's floating point arithmetic operations to be performed with two word (32-bit) precision. $R4 causes four word (64-bit) precision. The default real size depends upon the particular PME that you are using (that is, if your PME runs four word reals, the default is four words). This directive must occur before the first symbol in a compilation that isn't a comment.

NOTE: If you attempt to run a code file with one real size using a system configured for another real size, you will receive execution error 17 (real size mismatch).

$T - Title

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

$U - Use Library

There are two options indicated by $U. One is a string option (Use Library). The other, described below, is a toggle switch option (User Program).

With the Use Library option, the string is interpreted as a file name. This file should contain the unit(s) that your program is about to use. If the file is found, the compiler attempts to locate the unit(s) that it needs for the subsequent USES declarations. If a particular unit isn't found there, the compiler looks in *SYSTEM.LIBRARY.

If a client (program or unit) contains USES declarations but no $U option, the compiler looks for the used units first in the source file itself, and then in *SYSTEM.LIBRARY.

The following is an example of a valid USES clause using the $U option:

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

$U - User Program

This option is used to specify whether the compilation is your compilation, or a p-System compilation. If present, it must appear before the heading (that is, before the reserved word PROGRAM or UNIT).

When the default $U+ is in effect, your program is indicated. The $U- option allows system programmers to compile units with names that are predeclared in the p-System. These units are actually part of the p-System, itself. $U- also sets $R- and $I-.

In general you should never use this option, unless you need to compile GOTOXY (see the Adaptable System Installation Manual).

Conditional Compilation

You may conditionally compile portions of the source text. At the beginning of a program's text you can set a compile-time flag which determines whether or not the conditionally compiled text will be compiled.

In order to designate a section of text as conditionally compilable, you must delimit it by the options $B (for begin) and $E (for end). Both of these options must name the flag which determines whether the code between them is compiled. The flag itself is declared by a $D option at the beginning of the source. $D options may be used at other locations in the source to change the value of an existing flag.

Here is an example:

```
{$D DEBUG}  {declares DEBUG and sets it TRUE}
PROGRAM SIMPLE;
...
BEGIN
   ...
   {$B DEBUG}  {if DEBUG is TRUE, this section is compiled}
   WRITELN('There is a bug.');
   {$E DEBUG}  {this ends the section}
   ...
   ...
   {$B DEBUG-}  {if DEBUG is FALSE, this section is compiled}
   WRITELN('Nothing has failed.');
   {$E DEBUG}
   ...
END {SIMPLE}.
```

Each flag in a program must appear in a $D option before the source heading. The name of a flag follows the rules for Pascal identifiers. If the flag's name is followed by a '-', that flag is set false. The flag may be followed by a '+', which sets it true. If no sign is present, a flag is true. The flag's name may also be followed by a '^' as shown below.

The state of a flag may be changed by a $D option which appears after the source heading, but the flag must have first been declared before the heading.

The $B and $E options delimit a section of code to be conditionally compiled. The $B option may follow the flag's name with a '-', which causes the delimited code to be compiled if the flag is false. In the absence of a '-', the code is compiled if the flag is true. The flag's name may also be followed by a '+' or '^'; these are ignored. In a $E option, the flag's name may be followed by a '+', '-', or '^'; these symbols are ignored.

The state of each flag is saved in a stack, just as the state of a stack switch option is saved. Thus, using a $D option with '^' yields the previous value of the flag. Each flag's stack may be as many as 15 values deep. If a 16th value is pushed, the bottom of the stack is lost. If an empty stack is popped with '^', the value returned is always false.

If a section of code isn't compiled, any pseudo-comments it may contain are ignored as well.

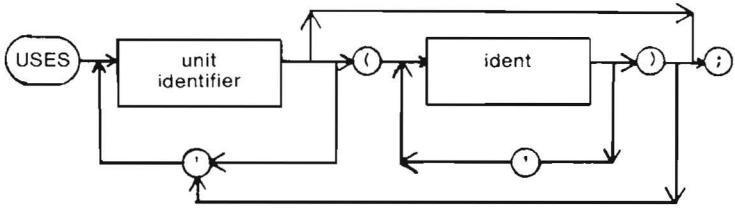
```
{$D DEBUG-}  {declares DEBUG and sets it FALSE}
PROGRAM SIMPLE;
...
BEGIN
   {$D DEBUG+}  {changes DEBUG to TRUE}
   ...
   {$B DEBUG}  {if DEBUG is TRUE, this section is compiled}
   WRITELN('There is a bug.');
   {$E DEBUG}  {this ends the section}
   ...
   ...
   {$D DEBUG                  }  {restores previous value of DEBUG}
                {... in this case, FALSE}
   {$B DEBUG-}  {if DEBUG is FALSE, this section is compiled}
   WRITELN('Nothing has failed.');
   {$E DEBUG}
   ...
END {SIMPLE}.
```

Selective Uses

Selective uses allows your programs to choose the items that you wish to use from a unit's interface section. You can often take advantage of this to reduce compile-time space requirements. Also, compilation time can be reduced. Both of these are especially noticeable when you are using units with large interface sections from which you only require a few items. This is because the rest of the interface section doesn't need to be compiled.

Also, selective uses is valuable for documentation purposes in that you can easily see the specific items that a client needs from the unit it uses.

The following diagram explains the syntax of selective uses:



In this diagram, ident can be a constant, type, variable, or routine (procedure, process, or function). Here is an example of a selective uses statement:

```
USES MYUNIT (A_CONST, VAR1, VAR2, MY_ROUTINE);
```

If a selected declaration isn't present in the interface text, an error results during compilation.

Any constant or type used in a selected declaration must be included in the selective uses list. For example, if VAR1 is of type TYPE1, the list above isn't acceptable unless TYPE1 is added (even though TYPE1 may not be directly required by the client being compiled).

You should list only the name of a routine. No explicit listing of parameters is needed. However, any types or constants that the parameters use must be explicitly included.

Most identifiers must be named explicitly in the identifier list if they are to be made available to the compiled module. Identifiers are available implicitly in these situations:

- 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).

Here is an example of selective uses. Unit A is selectively used by Units B and C.

```
Unit A;
interface
const
   maxnum=1000;
   maxchar=7;

type
   byte=0..255;
   codeblock=packed array
     [0..maxnum] of byte;
   alpha=packed array
     [0..maxchar] of char;
   ptr_info_rec=                              info_rec;
   info_rec=record
              code:codeblock;
              llink,rlink:ptr_info_rec;
            end;
    next=char;

var
   first,last:byte;

function update(var info:ptr_info_rec)
  :next;

implementation

  function update;
  begin
    with info                                   do
    begin
      llink:=rlink;
      if rlink=llink then
        update:='y'
      else
        update:='n';
    end;
  end;

end. {unit A}
```

```
Unit B;
interface
{$U a.code}
uses a( {const} maxchar,
  {include for type ALPHA}
        {types} alpha,
  {include for variable WHICH}
                byte,
  {include for FIRST and LAST}
        {vars } first,
  {include for proc CHANGE}
                last
  {include for proc CHANGE}

sing A

  maxchar=7;
  byte=0..255;
  alpha=packed array
    [0..maxchar] of char;
   first,last:byte;
      );

procedure change(which:alpha);

implementation

   procedure change;
   begin
     if which='        ' then
       last:=first
     else
       first:=last;
   end;

end; {unit B}
```

```
Unit C;
interface
implementation
{$U a.code}
uses a( {const} maxnum,
  {include for type CODEBLOCK}
                maxchar,
  {include for type ALPHA}
                byte,
  {include for type CODEBLOCK}
        {type}  alpha,
  {include for variable MINE}
                info_rec,
  {include for PTR_INFO_REC}
                ptr_info_rec,
  {include for func UPDATE}
                codeblock,
  {include for INFO_REC}
                next,
  {include for func UPDATE}
sing A
  maxnum=1000;
  maxchar=7;
  byte=0..255;
  codeblock=packed array
    [0..maxnum] of byte;
  alpha=packed array
    [0..maxchar] of char;
  ptr_info_rec=                              info_rec;
  info_rec=record
         code:codeblock;
         llink,rlink:ptr_info_rec;
           end;
    next=char;
 function update
   (var info:ptr_info_rec):next;
        {func}  update),
Using B
      b;

 var
    info:ptr_info_rec;
    mine:alpha;
 begin
   new(info);
   new(info                                    .rlink);
   info                                      .llink:=nil;
   mine:='newsystm';
   if update(info)='y' then

     writeln('info updated')
   else
     change(mine);
 end.
```

SEGMENTS, UNITS AND LIBRARIES

Segments, units, and libraries are three major facilities that help you manage large programs and effectively use main memory. These facilities enable very large programs to be developed in a microsystem environment; in fact, these facilities were used extensively in developing the system, itself.

Segmenting a Program

An entire program need not to be in main memory at run-time. Most programs can be described in terms of a working set of code that is required over a given time period. For most—if not all—of a program's execution time, the working set is a subset of the entire program, sometimes a very small subset. Portions of a program that aren't part of the working set can reside on disk, thus freeing main memory for other uses.

When the p-System executes a code file, it reads code into main memory. When the code has finished running, or the space it occupies is needed for some action having higher priority, the space it occupies may be overwritten with new code or new data. Code is swapped into main memory a segment at a time.

In its simplest form, a code segment includes a main program and all of its routines. A routine may occupy a segment of its own; this is accomplished by declaring it a segment routine. Segment routines may be swapped independently of the main program; declaring a routine a segment is useful in managing main memory.

Routines that aren't part of a program's main working set are prime candidates for occupying their own segment. Such routines include initialization and wrap-up procedures and routines that are used only once or only rarely while a program is executing. Reading a procedure in from disk before it is executed takes time. Therefore, the way that you divide up a program is important.

UCSD Pascal, FORTRAN, and BASIC use their own syntax for creating separate segments. Refer to each particular language's manual for more information on this.

Separate Compilation — Units

Separate compilation is a technique whereby segments of a program are compiled separately and subsequently executed as a coordinated whole.

Many programs are too large to compile within the memory confines of a particular microcomputer. Such programs might comfortably run on the same machine, especially if they are segmented properly. Compiling small pieces of a program separately can overcome this memory problem.

Separate compilation also allows small portions of a program to be changed without necessarily affecting the rest of the code. This saves time and is less error prone. Libraries of correct routines may be built up and used in developing other programs. This capability is important if a large program is being developed and is invaluable if the project involves several programmers.

These considerations also apply to assembly language programs. Large assembly programs (such as p-machine emulators) can often be more effectively maintained in several separate pieces. When all these pieces have been assembled, the L(inker puts them together and installs the linkages that allow the various pieces to reference each other and function as a unified whole.

You may also want to reference an assembly language routine from a higher-level language host program; for example, Pascal or FORTRAN. This may be necessary for performance reasons (assembly language is faster than p-code, the output of the compilers) or to provide low-level, machine-dependent or device-dependent handling.

Using the L(inker, the p-System allows assembly language routines to be linked with other assembly routines or into higher-level clients (programs or units). For more information about this, see the Assembler Reference Manual.

In UCSD Pascal, separate compilation is achieved by the unit construct—a unit being a group of routines and data structures. The contents of a unit usually relate to some common application, such as screen control or data file handling. A program or another unit may use the routines and data structures of a unit by simply naming it in a USES declaration. The term "host" refers to such a program, and "client compilation module" refers to a program or unit that uses another unit. In addition to being a separately compiled module, a unit is also a code segment, in that it can be swapped—as a whole—in and out of memory. You should note that it is possible for a unit's source text to be embedded in the client's source text if you don't want to compile a unit separately.

A unit consists of two main parts: (1) the interface section, which can declare constants, types, variables, procedures, processes, and functions which are public (available to any client module); and (2) the implementation section, in which private declarations can be made. These private declarations are available only within the unit and not to client modules.

Pascal, BASIC, and FORTRAN use their own syntax for separate compilation. (For more information about this, refer to each language's manual.)

Libraries

This section describes where you may place the code files that contain units so those units are available at compile-time or run-time. Run-time availability is described first.

There are four places where units may reside when the client's code is executed:

1. Within the client's code file.

2. In the SYSTEM.LIBRARY on the system disk.

3. In a user library.

4. In the operating system (SYSTEM.PASCAL).

The operating system units (described in the next chapter) are standard code. Don't place units that you write there. The other three options are available for units that you write or use.

In order to place a unit directly into a client's code file, use the Library utility, described in Chapter 6. Once the unit's code and the client's code are unified like this, the unit is available at run-time.

SYSTEM.LIBRARY generally contains standard units, such as the long integer package. You can add your units to this file with the Library utility. If you aren't currently using SYSTEM.LIBRARY, you can simply rename a unit's code file "SYSTEM.LIBRARY" and place it on the boot disk. Of course, you can add more files with the Library utility. All units that reside in SYSTEM.LIBRARY are available to clients.

A user library is any code file. The name of this code file must be in a "library text file." The standard default library text file is called USERLIB.TEXT and must be on the system disk. For example, if you create a USERLIB.TEXT containing these lines:

```
DISK2:SOME.UNITS
*MY.LIB
ANOTHER.CODE
```

These three code files are designated as user libraries. You don't have to specify the ".CODE" here. For example, the first file may be either DISK2:SOME.UNITS.CODE or DISK2:SOME.UNITS, depending upon which file is actually found. All three of these files may contain units which are then available for you to use.

When the p-System is searching several libraries for a unit, it first searches all of the user libraries in the order that they appear in the default library text file. It then searches *SYSTEM.LIBRARY. If you wish to include *SYSTEM.LIBRARY in the library text file, it is searched in the order that it appears. (If no library text file is used, only *SYSTEM.LIBRARY is searched.)

You can use a library text file, other then USERLIB.TEXT. Do this with the 'L' execution option. For example, if you select X(ecute from the Command menu and respond:

```
Execute what file? L=USERLIB2
```

During compile-time, as opposed to run-time, the code for a unit may reside in either of two locations:

1. *SYSTEM.LIBRARY

2. A code file specified in the text you are compiling.

Pascal, BASIC, and FORTRAN each have a way to indicate the names of units that are to be used. Each language also has a method for specifying the code files that contain those units. If you don't indicate a particular code file, the compilers search *SYSTEM.LIBRARY for any units you want to use. If you do indicate a code file, the compilers look there for the units. You can specify one unit as being in a particular code file, and another unit as being in a different code file if you wish.

GENERAL TACTICS

This section describes the use of segments and units. It presents a scenario for designing a large program, with some useful strategies.

Units and segments divide large programs into independent tasks. On microprocessor systems, the main bottlenecks in developing large programs are:

- A large number of variable declarations that consume space while a program is compiling.
- Large pieces of code that use up memory space while the program is executing.

Units address the first problem by: (1) allowing separate compilation; and (2) minimizing the number of variables needed to communicate between separate tasks. Segments alleviate the second problem by only requiring code that is in use to be in main memory at any given time; during this time, unused code resides on disk.

You can write a program with run-time memory management and separate compilations already planned, or you can write as a whole and then break it into segments and units. The latter approach is feasible when you're unsure about having to use segments or quite sure that they will be used only rarely. The former approach is preferred and easier to accomplish.

The following steps outline a typical procedure for constructing a relatively large application program:

1. Design the program (user and machine interfaces).

2. Determine needed additions to the library of units, both general and applied tools.

3. Write and debug units and add to libraries.

4. Code and debug the program.

5. Tune the program for better performance.

During design, try to use existing procedures to decrease coding time and increase reliability. You can accomplish this strategy by using units.

To determine segmentation, consider the expected execution sequence and try to group routines inside segments so that the segment routines are called as infrequently as possible.

While designing the program, consider the logical (functional) grouping of procedures into units. Besides making the compilation of a large program possible, this can help the program's conceptual design and make testing easier.

Units may contain segment routines within them. You should be aware that a unit occupies a segment of its own; except, possibly, for any segment routines it may contain. The unit's segment, like other code segments, remains disk-resident except when its routines are being called.

Steps 2 and 3 of the typical construction procedure are aimed at capturing some of the new routines in a form that allows them to be used in future programs. At this point, you should review, and perhaps modify, the design to identify those routines that may be useful in the future. In addition, useful routines might be made more general and put into libraries.

Program and test the Library routines before moving on to programming the rest of the program. This adds more generally useful procedures to the library.

The interface part of a unit should be completed before the implementation part, especially if several programmers are working on the same project.

Tuning a program usually involves performance tuning. Since segments offer greater memory space at reduced speed, performance is improved by: (1) turning routines into segment routines; or (2) turning segment routines back into normal routines. Either route is feasible. Pay some attention to the rules for declaring segments.

For information on languages, refer to the appropriate language manual.

CHAPTER 3

USER INTERFACE

INTRODUCTION

This chapter describes several facilities that can assist you in presenting a clean and portable user interface from your programs.

The first section describes run-time facilities that enable you to create your own applications environment. The p-System can run invisibly under your application using these facilities.

The next section describes the screen control unit. This unit, which is part of the operating system, can be used by your programs to easily handle the basic screen-oriented functions (such as clearing the screen, moving the cursor, and so forth).

Next, the error handler unit is covered. It enables your programs to intercept certain kinds of system errors and display your own messages. You might want to do this so that the error messages are specific to your particular application, or they are in a different language, and so forth.

After this, the command I/O unit is described. This unit allows you to redirect I/O and chain programs together. It is especially useful in conjunction with the run-time facilities in the first section.

RUN-TIME APPLICATION FACILITIES

As an applications developer, you may create programs which are automatically executed by the p-System. This exempts the end user from having to X(ecute these programs. The underlying p-System can even be completely hidden from such a user. You may present menus and prompts that apply specifically to your particular application.

If you name an executable code file, SYSTEM.STARTUP, and place it on the system disk, that program is executed when the p-System is booted. This program begins before the p-System's Command menu or welcome message is displayed.

SYSTEM.MENU operates similarly. It is executed each time the Command menu would normally be displayed.

Generally, SYSTEM.MENU is more useful for creating your own applications environments since it is called up repeatedly. Typically, you might place a simple menu-driven program in SYSTEM.MENU. This program displays the outer menus or prompts and services global issues related to your application package. When you select a component of your package, you would use the CHAIN procedure (within the operating system's Command I/O unit, described later in this chapter). CHAIN allows another program to be executed (without using the X(ecute command or displaying the Command menu in between). When that program completes its run, SYSTEM.MENU is again called. In this sort of scenario, the

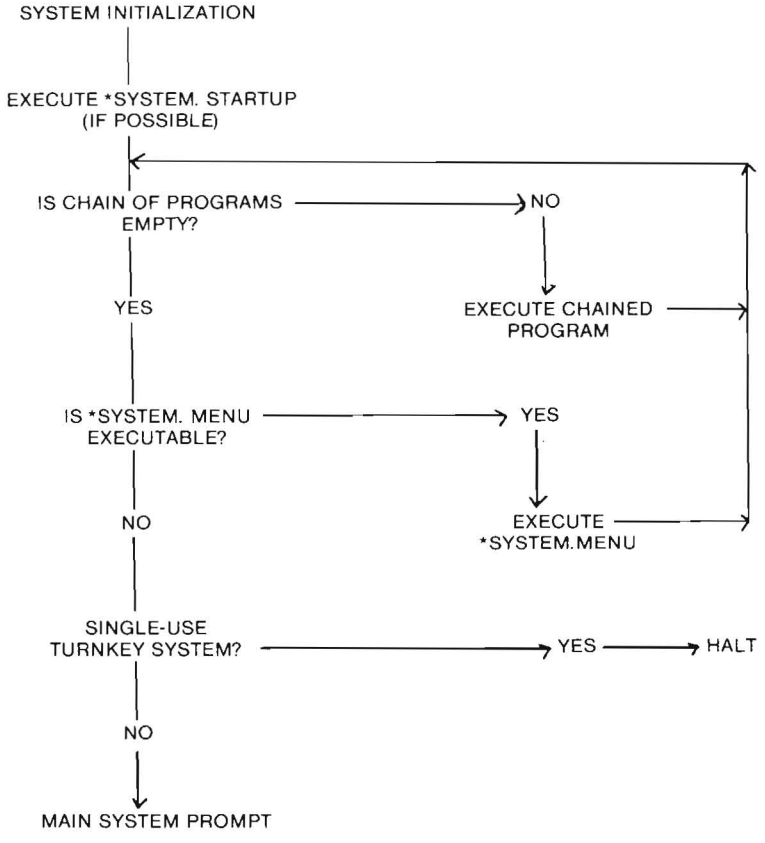
p-System's Command menu never appears.

Single-Use Run-Time System

The run-time system is a version of the p-System designed to package and execute a single application program or series of related programs called by the p-System chain mechanism just described. This version of the p-System never reaches the Command menu. The p-System components, such as the editor and filer, aren't a part of this package. The package may contain SYSTEM.STARTUP, but must contain SYSTEM.MENU.

System Initialization

The following diagram illustrates the flow of control each time the p-System is initialized:



THE SCREEN CONTROL UNIT

The screen control unit is a unit within the operating system which your programs can use to easily perform several useful screen-oriented tasks. These include blanking out a line or the entire screen, placing the cursor at a particular position, displaying p-System style menus, and so forth. These tasks are performed in a way that makes your programs transportable across different video displays.

You should realize that there is a special screen control unit for ANSI (American National Standards Institute) terminals. (These terminals use three character sequences. Most other terminals use, at most, two character sequences.) However, the interface section of this special version of the screen control unit is no different from the standard unit. This means that your programs don't have to be changed.

To use the screen control unit at compile time, you must have a copy of SCREENOPS.CODE with its interface section. A Pascal program must contain USES declaration similar to this:

```
USES {$U SCREENOPS.CODE} SCREENOPS;
```

At run-time, only the operating system needs to be available since it contains the SCREENOPS unit (only without the interface section).

SCREENOPS Interface Section

Here is a listing of the interface section for SCREENOPS:

```
unit screenops;

interface

const
  sc_fill_len = 11;
  sc_eol = 13;

type
  sc_chset        = set of char;
  sc_misc_rec     = packed record
                      height, width : 0..255;
                      can_break, slow, xy_crt, lc_crt,
                      can_upscroll, can_downscroll : boolean;
                    end;
  sc_date_rec     = packed record
                      month : 0..12;
                      day :   0..31;
                      year :  0..99;
                    end;
  sc_info_type    = packed record
                      sc_version : string;
                      sc_date : sc_date_rec;
                      spec_char : sc_chset; {Characters not to echo}
                      misc_info : sc_misc_rec;
                    end;
  sc_long_string  = string[255];
  sc_scrn_command = (sc_whome, sc_eras_s, sc_erase_eol, sc_clear_lne,
                     sc_clear_scn, sc_up_cursor, sc_down_cursor,
                     sc_left_cursor, sc_right_cursor);
  sc_key_command  = (sc_backspace_key, sc_dc1_key, sc_eof_key, sc_etx_key,
                     sc_escape_key, sc_del_key, sc_up_key, sc_down_key,
                     sc_left_key, sc_right_key, sc_not_legal, sc_insert_key,
                     sc_delete_key);
  sc_choice       = (sc_get, sc_give);
  sc_window       = packed array [0..0] of char;
  sc_tx_port      = record
                      row, col,              { screen relative}
                      height, width,         { size of txport (zero based)}
                      cur_x, cur_y   : integer;
                                {cursor positions relative to the txport }
                    end;

    {entries 4..syscom                .subsidstart-1 are valid}
  sc_err_msg_array = array [4..4] of          string;  {accessed $R-}

var
  sc_port : sc_tx_port;
  sc_printable_chars : sc_chset;
```

```
  sc_errorline : integer;
  sc_errormessage :                    sc_err_msg_array;

procedure sc_use_info(do_what:sc_choice; var t_info:sc_info_type);
procedure sc_use_port(do_what:sc_choice; var t_port:sc_tx_port);
procedure sc_erase_to_eol(x,line:integer);
procedure sc_left;
procedure sc_right;
procedure sc_up;
procedure sc_down;
procedure sc_getc_ch(var ch:char; return_on_match:sc_chset);
procedure sc_clr_screen;
procedure sc_clr_line (y:integer);
procedure sc_home;
procedure sc_eras_eos (x,line:integer);
procedure sc_goto_xy(x, line:integer);
procedure sc_clr_cur_line;
function  sc_find_x:integer;
function  sc_find_y:integer;
function  sc_scrn_has(what:sc_scrn_command):boolean;
function  sc_has_key(what:sc_key_command):boolean;
function  sc_map_crt_command(var k_ch:char):sc_key_command;
function  sc_prompt(line :sc_long_string; x_cursor,y_cursor,x_pos,
                    where:integer; return_on_match:sc_chset;
                    no_char_back:boolean; break_char:char):char;
function  sc_check_char(var buf:sc_window; var buf_index,bytes_left:integer)
                      :boolean;
function  sc_space_wait(flush:boolean):boolean;
procedure sc_init;
```

Routines within SCREENOPS

This section describes the routines within the screen control unit. The text ports mentioned here are rectangular portions of the screen that may be defined as smaller than the real screen. At present, this feature isn't fully implemented. Where text ports are mentioned in this section, the entire screen is the default.

Procedure SC_Init;

Usually, only the operating system calls this procedure. It initializes all the screen control tables and variables.

Procedure SC_Clr_Cur_Line;

Erases the current line.

Procedure SC_Clr_Line (Y: integer);

Clears line number Y within the current text port.

Procedure SC_Clr_Screen;

Clears the screen.

Procedure SC_Erase_to_EOL
(X, Line: integer);

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

Procedure SC_Eras_EOS
(X, Line: integer);

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

Procedure SC_Left;

Moves the cursor one character to the left.

Procedure SC_Right;

Moves the cursor one character to the right.

Procedure SC_Up;

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

Procedure SC_Down;

Moves the cursor one line down.

Procedure SC_Home;

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

Procedure SC_GOTO_XY
(X, Line: integer);

Moves the cursor to position (X, Line).

Function SC_Find_X: integer;

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

Funtion SC_Find_Y: integer;

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

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

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

Function SC_Space_Wait
(Flush: Boolean): Boolean;

This function repeatedly reads from the keyboard until a space or the ALTMODE character is received. Before doing this, it does a UNITCLEAR(1) if flush is true, and displays 'Type <space> to continue'. It returns true if a space wasn't read. After reading a <space> successfully, this function echoes a carriage return on the console.

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

This function displays the menu line (SC_Long_String is a STRING [255]) in the current text port at (X_Pos, Where). The cursor is placed at (X_Cursor, Y_Cursor) after the prompt is printed. If X_Cursor is less than 0, the cursor is placed at the end of the prompt. If the prompt is too large to fit within the current text port, it is broken up into several pieces, but only at the Break_Char. You can view different parts of the prompt (cycling through them) by entering '?'. If you only want to display the prompt, No_Char_Back should be true. In this case, SC_Prompt returns a function value of NUL, ASCII 0. If you want to receive a character from the user at the keyboard, No_Char_Back should be false. (In this case, SC_Prompt returns a functon value of the character received.) The keyboard is repeatedly read until the character read matches one within the Return_on_Match set. This set should be all capitals (for alphabetic characters) since your input is converted to uppercase when necessary.

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

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

Function SC_Map_CRT_Command
(VAR K_CH: char): SC_Key_Command;

SC_Key_Command is a type consisting of the following elements: SC_Backspace_Key, SC_DC1_Key, SC_EOF_Key, SC_ETX_Key, SC_Escape_Key, SC_DEL_Key, SC_Up_Key, SC_Down_Key, SC_Left_Key, SC_Right_Key, SC_Not_Legal. The character passed is mapped into one of these elements. SC_Not_Legal is where all characters are mapped which don't fit into one of the other ten categories. Prefix characters are recognized by this function. If you pass a prefix character, a nonechoed read is done to get the next character (before the mapping is performed). In this case, K_CH is returned as that character. For the ANSI version of Screenops, another read may be done (since three character codes are used on ANSI terminals).

Function SC_Scrn_Has
(What: SC_Scrn_Command): Boolean;

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

Function SC_Has_Key
(What: SC_Key_Command): Boolean;

SC_Key_Command consists of the elements previously listed in the description of SC_Map_CRT_Command. This function returns true if the keyboard generates the character passed.

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

This function is used to pass information back and forth between a program and the screen control unit. Do_What may either be SC_Get or SC_Give and indicates whether the program is getting information from the screen control unit or giving information to it. T_Info contains various items to be either passed or received. The following information is contained within T_Info.

```
SC_Version: string;
SC_Date: PACKED RECORD
           Month: 0..12;
           Day:   0..31;
           Year:  0..99;
         END;
Spec_Char: SET OF char; (* Characters not to echo *)
Misc_Info: PACKED RECORD
             Height, Width: 0..255;
             Can_Break, Slow, XY_CRT, LC_CRT,
             Can_UpScroll, Can_DownScroll: Boolean;
           END;
```

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

This function works like SC_Use_Info above. The contents of T_Port are either passed or received from the screen control unit. T_Port contains the following information.

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

ERROR HANDLER UNIT

Under certain circumstances, the p-System displays execution error messages. If a code segment is needed and the disk containing it isn't in the appropriate drive, you are asked to replace the disk and press <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 you are asked to press <space>, at which point the p-System is reinitialized.

When certain errors occur, your programs can alter the message that is displayed. This is useful for applications developers, especially those whose customers speak languages other than English.

Format of Error Messages

Error messages are displayed on one specified 80-column line. For example, when a code segment is needed from a disk that isn't present in the appropriate drive, the following prompt is displayed:

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

This indicates that the segment SEGNAME wasn't found on volume in device U. Place the volume VOLNAME in the correct drive and press <space>. Execution should continue normally.

The following example shows the error message that occurs when you press the p-System BREAK key.

```
Program Interrupted by user-Seg PASCALIO P# 17 0# 310 <space> continues
```

After <space> is pressed, the p-System is reinitialized.

System error messages, such as these, 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) is written to the console device when the message is written out.

After pressing <space>, the message line disappears; and, when possible, the cursor returns to its previous position. 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 lose the previous cursor position. This is because the p-System isn't informed of the cursor position after these kinds of low-level I/O operations.

User Control of Error Messages

Your 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 isn't 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 ERRORHANDLING unit provides these facilities. The file ERRORHANDL.CODE contains this unit. To use ERRORHANDLING, a Pascal program should have a declaration similar to the following example.

```
USES {$U ERRORHAND.CODE} ERRORHANDLING;
```

Also, ERRORHANDLING must be available at run-time, either in a library or placed into the using program's code file with the Library utility.

The following procedures are available within this unit:

```
Procedure Set_Error_Line
  (Line:Integer);

Procedure Set_User_Message
  (Drive:Integer; Mesg:String);
```

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

You may change the standard message for code segments needed on disks that aren't present. By calling SET_USER_MESSAGE with the DRIVE parameter set to the physical device number and the MESG parameter set to the desired message string.

Then, if a code segment is required from a missing disk in the unit for which your program has designated a special error message, that message is displayed. The p-System then waits for you to press <space>, whether or not your message actually indicates that a space is needed. The message line is subsequently erased; the cursor returns to its former position, if possible; and execution continues.

CAUTION: Your message is destroyed by a release if a MARK was 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.

For other kinds of execution errors, a standard p-System message is displayed on the message line. A fatal error always causes the p-System to fail. For nonfatal errors, the p-System waits for you to press <space>. The message line is then erased, the cursor returns to its former position, and execution continues (most likely the p-System reinitializes itself).

To proceed from a nonfatal error, press <esc>.

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

Error message values you set remain in effect during the program run, but are reset at program termination or whenever p-System reinitialization occurs.

Your 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.

THE COMMAND I/O UNIT

Command I/O is a unit in the operating system. From Pascal, your program should contain the statement:

```
USES {$U commandio.code} COMMANDIO;
```

Then, the following procedures are available to the program:

Procedure Chain
(Exec_Options: String);

A call to CHAIN causes the system to X(ecute EXEC_OPTIONS after the calling program (the chaining program) has terminated. The effect is that of: (1) manually pressing 'X' to call X(ecute; and (2) entering the characters in EXEC_OPTIONS. Neither the Command menu nor the X(ecute prompt is displayed; the system goes on to immediately perform the actions indicated by EXEC_OPTIONS.

If a program (or sequence of programs) contains more than one call to CHAIN, the EXEC_OPTIONS are saved in a queue and performed on a first-in-first-out basis before returning control of the system to you.

To clear the queue, call CHAIN with an empty string (for example, "CHAIN('');").

An execution error or an error in an EXEC_OPTIONS string clears the queue, returning control to you. A call to EXCEPTION, described below, may also clear the queue.

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

Function Redirect
(Exec_Options: String) : Boolean;

This should contain only option specifications and not the name of a file to execute (to execute a program from another program, see the CHAIN intrinsic).

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

If an error occurs during a call to REDIRECT, the state of redirection is indeterminate; this is a dangerous condition. If REDIRECT returns false, your program should follow it with a call to EXCEPTION, in order to turnoff all redirection. If you don't do this, the results are unpredictable.

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

Procedure Exception
(Stopchaining: Boolean);

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

Whenever an execution error occurs, an EXCEPTION(TRUE) call is made (leaving redirection on after an error leaves the system in an indeterminate state).

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

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 you to save figures in disk files and retrieve them.

The simplest Turtlegraphics routines are intentionally very easy to learn and use. Once you are 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 you 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 you (it doesn't 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 you. This allows your own coordinates to be mapped into pixels on the display. All other figures are scaled by the global display scale.

You 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 you 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 is accessible from FORTRAN and BASIC. This is described later in this section.

The Turtle

The turtle is an imaginary creature in the display screen that will draw lines as you move 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 isn't 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, you 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 isn't 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; refer to "Installing Turtlegraphics," below.

A sample set of colors might be:

0 = Black
1 = Blue
2 = Red
3 = Magenta
4 = Green
5 = Cyan
6 = Yellow
7 = White

Turtlegraphics uses a numeric designation for color instead of a symbolic designation like the word blue or red to maintain the p-System language and hardware compatibility. For example, while Pascal would allow the use of symbolic color designations, BASIC and FORTRAN wouldn't.

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 doesn't change until Pen_Mode is called again.

These are the possible modes:

0 Nop - doesn't 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 isn't 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-hand corner of the screen, ready to move right.

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 you 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 isn't possible for the display to return to its original state, and the p-System software has no knowledge of what that original state was. Also, for the system to operate correctly, you must follow any video mode change with a call to Display_Scale.

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.

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 "Installing Turtlegraphics," below, 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.

Scaling

When you wish 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 your 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 you require, 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 user-written procedure Query_Environment. (Refer to "Installing Turtlegraphics," below.)

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 you wish 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 (for example, circles, squares, pie-charts, and so forth), it must ensure 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 doesn't alter the pixel position of any of the turtles in the figures. However, the values returned from X_Pos and Y_Pos may change.

Figures and the Port

You 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 (Get_Figure).

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. You might create a figure, specify the port, then display that figure (or a portion of it) within the port. Specifying a viewport doesn't 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 constructs may rapidly exhaust the memory available in the p-System due to heap fragmentation. For example, a figure may be created using Create_Figure (or it may be read in from disk using Function Load_Figure, described below). If possible, after that figure is used (for example, with a 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-hand corner is at (Corner_X, Corner_Y). The X and Y positions are specified in the units of the current display scale. If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

The following items define the drawing mode numbers.

0 Nop - Doesn't 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 isn't of the source's background color replaces the corresponding pixel in the destination.

3 Underwrite - Each pixel in the source that isn't 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 isn't 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 you wish 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 won't be displayed (this is true even if the window is subsequently expanded to encompass a previously drawn line). The viewport boundaries are specified in the units of the current display scale. If the specified size of the viewport is larger than the current range of the display, the viewport is truncated to the display limits.

Pixels

It is possible to ascertain (Read_Pixel) or 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 you 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: Integ

Sets the pixel at the X,Y location of the specified figure to the specified color. The X,Y location is specified in the units of the current Display_Scale.

Fotofiles

You 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 isn't 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 didn't successfully store the figure.

Routine Parameters

The following 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:integer );
    procedure Get_Figure( source_screen:
                                    integer;
                                    corner_x, corner_y: real;
                                    copymode : 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( direction : real );
    procedure Pen_Mode( state : integer );
    procedure Pen_Color( shade : integer );
    procedure WChar( c: char; copymode, shade: integer );
    procedure WString( s: string; copymode, shade: integer);
```

Sample Program

Here is a sample program that illustrates a number of Turegraphics 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);
      {respecify viewport in
       the lower left-hand corner}
    GetFigure (I, 0, 0, 1);
      {display finished spiral}
    readln;
      {clear user input buffer}
end.
```

Using Turtlegraphics from FORTRAN

Using the Turtlegraphics routines from FORTRAN requires accessing special interface units at compile time. This is because the Pascal syntax contained in the standard Turtlegraphics unit isn't compatible with FORTRAN. In order to use Turtlegraphics from FORTRAN, the FORTRAN source program must contain a directive similar to this:

```
$USES TURTLEGRAPHICS IN FTN.TURTLE.CODE
```

The Turtlegraphics unit in FTN.TURTLE.CODE contains the special interface section that is FORTRAN compatible. It is accessed at compile time only. During program execution, the Turtlegraphics unit in SYSTEM.LIBRARY is accessed automatically.

In addition, there are two functions, Readfigurefile and Writefigurefile, and one procedure, Wstring, that can't be directly referenced from FORTRAN. Instead, a separate unit called FTURTLEGRAPHICS exists to provide the support necessary to pass FORTRAN arguments to these three routines. In order to call Readfigurefile, Writefigurefile or Wstring, a directive similar to this:

```
$USES FTURTLEGRAPHICS IN RTLIB4.CODE
```

must appear in the FORTRAN program. RTLIB4.CODE refers to the name of the FORTRAN run-time library.

In order to call the Turtlegraphics routines from FORTRAN, the following general guidelines should be obeyed:

- FORTRAN allows identifiers to be a maximum of six characters only. PASCAL routines with longer names need to be truncated when they are called from FORTRAN.

- PASCAL boolean variables are referred to as logical in FORTRAN.

- FORTRAN refers to procedures as subroutines. The word CALL must precede the subroutine name in FORTRAN to indicate a subroutine or procedure call.

The remainder of this section describes the parameters of the routines using the appropriate FORTRAN syntax.

```
SUBROUTINE MOVE ( DISTANCE )
REAL DISTANCE

SUBROUTINE MOVETO ( X, Y )
REAL X, Y

SUBROUTINE TURN ( ROTATI )
REAL ROTATI

SUBROUTINE TURNTO ( HEADIN )
REAL HEADIN

SUBROUTINE PENCOL ( SHADE )
INTEGER SHADE

SUBROUTINE PENMOD ( MODE )
INTEGER MODE

REAL FUNCTION TURTLX ( )

REAL FUNCTION TURTLY ( )

REAL FUNCTION TURTLA ( )

SUBROUTINE ACTIVA ( SCREEN )
INTEGER SCREEN

SUBROUTINE FILLSC ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

SUBROUTINE BACKGR ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

SUBROUTINE DISPLA ( MINX, MINY, MAXX, MAXY )
REAL MINX, MINY, MAXX, MAXY

REAL FUNCTION ASPECT ( )

INTEGER FUNCTION CREATE ( XSIZE, YSIZE )
REAL XSIZE, YSIZE

SUBROUTINE DELETE ( SCREEN )
INTEGER SCREEN

SUBROUTINE GETFIG ( SOURCE, XCOR, YCOR, MODE )
INTEGER SOURCE, MODE
REAL XCOR, YCOR

SUBROUTINE PUTFIG ( DESTIN, XCOR, YCOR, MODE )
INTEGER DESTIN, MODE
REAL XCOR, YCOR

INTEGER FUNCTION READPI ( SCREEN, X, Y )
INTEGER SCREEN
REAL X, Y
```

```
SUBROUTINE SETPIX ( SCREEN, X, Y, SHADE )
INTEGER SCREEN, SHADE
REAL X, Y

INTEGER FUNCTION LOADFI ( INDEX )
INTEGER INDEX

INTEGER FUNCTION STOREF ( FIGURE )
INTEGER FIGURE

SUBROUTINE WCHAR ( C, COPYMODE, SHADE )
CHARACTER*1 C
INTEGER COPYMODE, SHADE
```

The following three routines are contained in the FORTRAN interface unit, FTURTLEGRAPHICS, in the FORTRAN run-time library. Because a character string can't be passed directly to the functions Readfigurefile and Writefigurefile and the procedure Wstring, the FORTRAN routines FREADF, FWRITE and FWSTRING must be called instead.

For the functions FREADF and FWRITE, TITLE refers to the name of the Fotofile which is to be read or written and LEN contains the length of the TITLE variable.

```
INTEGER FUNCTION FREADF ( TITLE, LEN )
CHARACTER*N TITLE
INTEGER LEN

INTEGER FUNCTION FWRITE ( TITLE, LEN )
CHARACTER*N TITLE
INTEGER LEN
```

The following FORTRAN statements are necessary to call them:

```
CHARACTER * 5  TITLE
INTEGER LEN, IRESLT

LEN = 5
TITLE = 'FOTO1'
IRESLT = FWRITE ( TITLE, LEN )

or

IRESLT = FREADF ( TITLE, LEN )
```

For FWSTRING, C is actually a string of length LEN which is to be written in mode COPYMODE with shade SHADE.

```
SUBROUTINE FWSTRING ( C, LEN, COPYMODE, SHADE )
CHARACTER * <length of string> C
INTEGER LEN, COPYMODE, SHADE
```

To call FWSTRING, the following FORTRAN statements are necessary:

```
LEN = <length of string>
C   = <string>
CALL FWSTRING ( C, LEN, COPYMODE, SHADE )
```

Using Turtlegraphics From BASIC

Using the Turtlegraphics routines from BASIC requires accessing a special interface unit at compile time. This is because the Pascal syntax contained in the standard Turtlegraphics unit isn't compatible with BASIC. In order to use Turtlegraphics from BASIC, the BASIC source program must contain directives similar to these:

```
LIBRARY "BSC.TURTLE.CODE"
USES TURTLEGRAPHICS
```

The Turtlegraphics unit in BSC.TURTLE.CODE contains the special interface section that is BASIC compatible. It is accessed at compile time only. During program execution, the standard Turtlegraphics unit in SYSTEM.LIBRARY is accessed automatically.

In addition, the procedure Wchar contains a parameter of type CHAR which was changed to type INTEGER for BASIC, when calling this procedure the DIM var*1 variable in BASIC must be changed to an INTEGER.

In order to call the Turtlegraphics routines from BASIC, the following general guidelines should be obeyed:

- BASIC doesn't allow imbedded reserved words in identifiers, no identifiers should have this characteristic.

- BASIC refers to procedures as subroutines. The word CALL must precede the subroutine name in BASIC to indicate a subroutine or procedure call.

The remainder of this section describes the parameters of the routines using the appropriate BASIC syntax.

```
SUB DSPSCALE ( MINX, MINY, MAXX, MAXY )
REAL MINX, MINY, MAXX, MAXY

DEF REAL ASPECTRATIO

DEF INTEGER CREADFIGURE ( XSIZE, YSIZE )
REAL XSIZE, YSIZE

SUB DELETEFIGURE ( SCREEN )
INTEGER SCREEN

SUB VIEWPORT ( MINX, MINY, MAXX, MAXY )
REAL MINX, MINY, MAXX, MAXY

SUB FILLSCREEN ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

SUB BACKGROUND ( SCREEN, SHADE )
INTEGER SCREEN, SHADE

DEF INTEGER RDPIXEL ( SCREEN, X, Y )
INTEGER SCREEN
REAL X, Y

SUB SETPIXEL ( SCREEN, X, Y, SHADE )
INTEGER SCREEN, SHADE
REAL X, Y

SUB GETFIGURE ( SOURCESCREEN, CORNERX, CORNERY, COPYMODE )
INTEGER SOURCESCREEN, COPYMODE
REAL CORNERX, CORNERY

SUB PUTFIGURE ( DESTINATIONSCREEN, CORNERX, CORNERY, COPYMODE)
INTEGER DESTINATIONSCREEN, COPYMODE
REAL CORNERX, CORNERY

DEF INTEGER RDFIGURE ( TITLE )
DIM TITLE$*8

DEF INTEGER WRTFIGURE ( TITLE )
DIM TITLE$*8

DEF INTEGER LDFIGURE ( INDEX )
INTEGER INDEX

DEF INTEGER STORFIGURE ( SCREEN )
INTEGER SCREEN

SUB ACTIVATETURTLE ( SCREEN )
INTEGER SCREEN

DEF REAL TURTLX

DEF REAL TURTLY

DEF REAL TURTLANGLE
```

```
SUB MOVE ( DISTANCE )
REAL DISTANCE

SUB MOVETO ( X, Y )
REAL X, Y

SUB TURN ( ROTATION )
REAL ROTATION

SUB TURNTO ( DIRECTION )
REAL DIRECTION

SUB PENMODE ( STATE )
INTEGER STATE

SUB PENCOLOR ( SHADE )
INTEGER SHADE

SUB WCHR ( C, COPYMODE, SHADE )
INTEGER C, COPYMODE, SHADE
```

NOTE: To call procedure, WCHR the following instructions must be used:

```
INTEGER C
DIM S$*1

C = ASC(S) (* S contians the character to print*)
CALL WCHR ( C, COPYMODE, SHADE )

SUB WSTR ( S, COPYMODE, SHADE )
DIM S$
INTEGER COPYMODE, SHADE
```

Installing Turtlegraphics

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 seven 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 set 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 aren't already familiar with Turtlegraphics, you should stop and read its description in Chapter 1 of this manual for your particular hardware. The following section, entitled "Graphics I/O Routines," explains these low-level routines and the structures they manage. It also provides some implementation hints 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. The "Graphics System Initialization" section, presented later in this chapter, describes how to tailor the supplied unit to suit your requirements. If your system requires no special configuration to perform graphics I/O, skip the "Graphics System Initialization" section, 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. A subsequent section, "Character Fonts," describes the structure of *SYSTEM.FONT and how to build a custom version. It is strongly recommended that you don't 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. "Linking and Librarying," below, 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.

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 identifie
                          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, Draw_Line is NOT called.

Substitute mode calls for every visible point on the line segment to be unconditionally plotted.

Overwrite, for the purposes of Draw_Line, 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_xy;
       {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 (x, y);
                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;
```

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 is 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.

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] ofBoolean
```

Don't 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 isn't evenly divisible by 16.

Linking and Librarying

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. Select either GRAFIX2.CODE or GRAFIX4.CODE to host your linking, depending on the real number size of your p-machine. The output file should be called TURTLE.CODE.

NOTE: There are least-significant-byte-first and most-significant-byte-first versions of these GRAFIXx files. You should have received the Turtlegraphics package which corresponds to the byte sex of your processor. It isn't possible to link assembled routines into a host of the opposite byte sex.

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.

Display Set and Clear Pixel Test

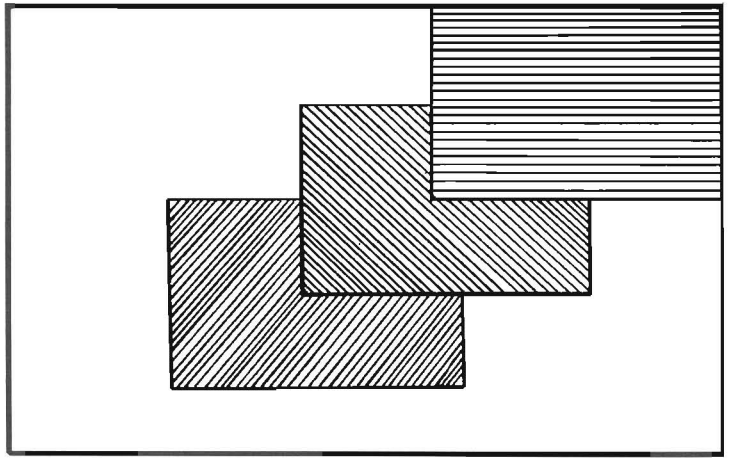
This test should display a set of colored, dotted lines horizontally across the display, drawing a pair from left-to-right with each pass. The test should end when it has cycled through all the colors available on your display. It will then query you to see that it has determined the number of available colors correctly. For this and all subsequent queries, affirm a correct result by pressing 'Y' (either upper- or lowercase). Any other character is considered a negative response, and the exercises will terminate.

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.

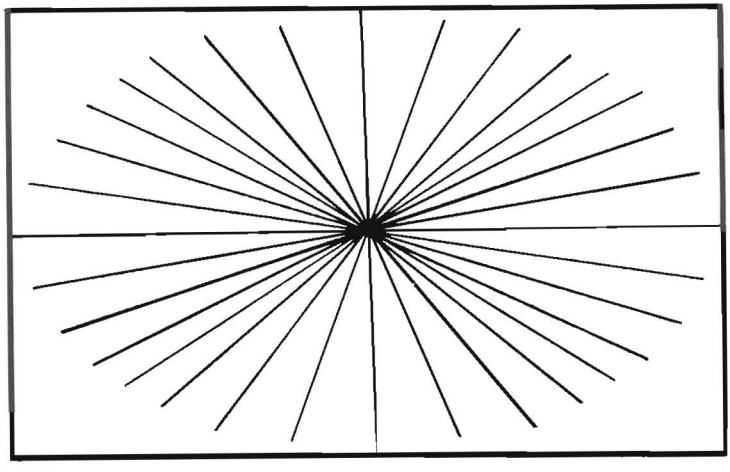
The next phase of this test should set the screen to color 0 and then display a set of overlapping rectangles from the lower left-hand corner of the display to the upper right, using all the available colors. This is a test of windowing in Fill_Color.

The screen for a 4-color system is shown below:

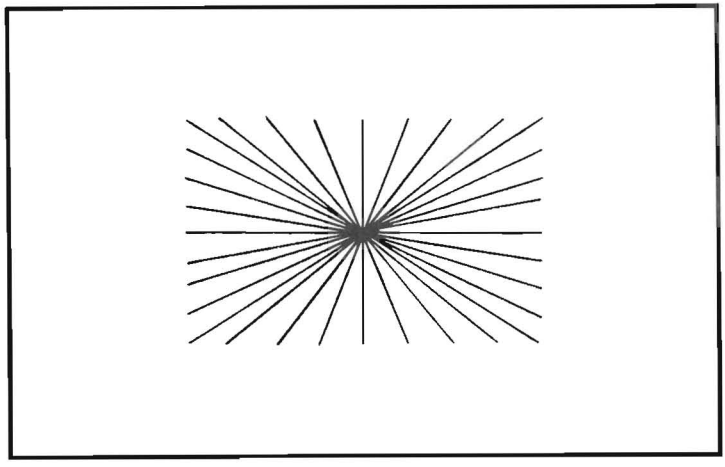


Display Line-Drawing Exercises

The next portion of the exercises is designed to check the Draw_Line routine. First a set of radial lines are drawn from the center of the screen. Thirty-six radials are drawn starting at the 3 o'clock position, and then move counterclockwise around the center point. This behavior is repeated for all the nonzero 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 checks 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 shows how the display should appear when the test is 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:

0 In Nop mode nothing else should appear.

1 Substitute mode should draw lines over the top of the rectangle and beyond.

2 Overwrite mode should draw lines over the top of the rectangle and beyond.

3 Underwrite mode shouldn't alter the center rectangle. Radials should be visible from the periphery of the rectangle, continuing out to their previous end point.

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. Your 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 can't contain a viewport.

We remind you that a test can prove the presence of bugs, but never their absence. EXERCISE won't 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.

QUICKSTART Units

The code files PEDGEN.CODE and CHKSUMOPS.CODE are related to the QUICKSTART utility (described in the Operating System Reference Manual). These are two standard p-System units which may be used by p-System programs in order to perform tasks related to the quickstarting of programs.

PEDGEN contains a single routine called PED_GENERATE. PED_GENERATE creates a new code file which contains a description of the execution environment required by the program. This execution environment description is in the form of a "Program Environment Descriptor" referred to as a "PED." The Pascal interface to this unit is described below.

CHKSUMOPS contains routines to generate and validate checksums for p-System code files. The Pascal interface to this unit is described after PEDGEN.

PEDGEN Unit Interface

Here is the Pascal interface to the PEDGEN unit:

```
unit pedgen;

  {This unit contains the standard p-System routine which
   installs a new Program Execution Environment Descriptor
   (PED) into a program code file.}

 interface

   const max_pedgen_file_name_length = 255;

   type pedgen_file_name = string[max_pedgen_file_name_length];

   pedgen_result =  {Result codes returned by
                     PED_GENERATE.}

          (pgr_no_error,
                   {Result indicating
                    successful
                    operation.}

           pgr_lib_error,
                   {Indicates I/O error either
                    on open or read of a
                    library code file.}

           pgr_lib_output_error,
                   {Indicates I/O error when
                    creating a copy of an updated
                    library code file.}

           pgr_chksum_error,
                   {I/O error occurred when
                    attempting to insert new
                    checksum into a referenced
                    library code file.}

           pgr_input_error,
                   {Indicates I/O error either
                    on open or read of host
                    program code file.}

           pgr_output_error,
                   {Indicates I/O error writing
                    PED to disk file.}

           pgr_unit_error,
                   {Indicates failure to locate
                    a referenced unit.}

           pgr_bad_library_list_error,
                   {Library file list text file
                    is not a text file.}

           pgr_lib_list_error,
                   {Indicates I/O error reading
                    library file list text file.}
```

```
        pgr_duplicate_unit_error,
                {A unit name conflicts with
                 a system unit name, or the
                 system contains more than
                 one unit with the same name.}

        pgr_lib_count_error,
                {Number of library files referenced
                 by execution environment exceeds
                 max_library_file_refs.}

        pgr_sys_ref_count_error,
                {Number of system segments referenced
                 by execution environment exceeds
                 max_system_seg_refs.}

        pgr_no_program_error,
                {Input file is not a host
                 program, or the operating system
                 host unit is missing from an
                 operating system host code file.}

        pgr_no_boot_seg_error,
                {System host code file does not
                 contain the required boot segment.}

        pgr_must_be_linked_error,
                {Program environment references
                 a segment which contains
                 unresolved references to
                 assembly language routines.
                 Thus the program must be
                 linked by the Linker before an
                 environment can be constructed.}

        pgr_obsolete_segment_error,
                {Program contains a reference to
                 a segment which was not compiled
                 with a Version IV compiler.}

        pgr_not_enough_mem_error,
                {Not enough memory to build
                 required temporary data
                 structures during environment
                 construction process.}

        pgr_buf_overflow_error
                {The buffer into which the PED
                 is being generated in not large
                 enough to describe the environment
                 for the program.}

        );


{The following is the interface to the PED_GENERATE routine itself.}

function
  ped_generate
    (input_file_id: pedgen_file_name;
                {File name of program code file for
                 which a new PED is to be constructed.}

     output_file_id: pedgen_file_name;
                {File name of new code file to be
                 created.}
```

```
is_system: boolean;
          {If TRUE the PED for a new operating
           system is to be constructed which does
           not contain references to segments of
           the current operating system.}

copy_input: boolean;
          {If TRUE the PED is inserted in a new copy
           of the source code file; otherwise the new
           PED is written to the original code file.}

copy_libraries: boolean;
          {Controls whether user is prompted for
           where to copy updated versions of library
           code files into which new checksums have
           been inserted.}

write_progress_messages: boolean;
          {If TRUE progress messages are written
           to the standard file OUTPUT describing
           how the execution environment is being
           constructed.}

var the_iorslt: integer;
          {When an I/O result is returned this
           parameter is set to the value of IORESULT.
           If no I/O errors occur, this is set to zero.}

var the_name: pedgen_file_name
          {When a unit or a library code file is
           not found, or an I/O error occurs this
           variable is set to the name of the unit
           or file.  When none of these errors
           occur, this variable is set to the
           empty string.}

): pedgen_result;
```

The INPUT_FILE_ID parameter specifies the file name of the code file for which a Program Environment Descriptor (PED) should be constructed. The OUTPUT_FILE_ID parameter specifies the file name for the new code file to be created. OUTPUT_FILE_ID is only used when the COPY_INPUT parameter has the value TRUE.

The IS_SYSTEM parameter is used to determine if the program is a new version of the p-System operating system. If IS_SYSTEM is TRUE, PED_GENERATE generates a PED that doesn't contain any information specific to the currently executing operating system. If IS_SYSTEM is FALSE, the generated PED assumes the current operating system environment.

The COPY_INPUT parameter specifies whether the PED is to be installed in the existing code file, or installed into a copy of the original program code file. If COPY_INPUT is TRUE, a copy of the original program code file is written to the file specified by the OUTPUT_FILE_ID parameter; otherwise, the source code file is modified to contain a new PED.

This copying process begins by copying the segment dictionary blocks of the original code file to the designated output file. The segments contained in the original code file are copied one at a time to the output file. When all of the code segments within the original code file have been copied, a revised segment dictionary is created in a sequence of consequtive blocks at the beginning of the new code file. The manner in which the contents of the original code file are transferred to the output code file ensures that a PED present in the original code file doesn't occupy space in the new program code file. Once this copying process is completed, the building of a new PED for the program is started. This final process consists of building the execution environment for the program and storing a representation of that

environment in the form of a PED in the new code file. The PED is stored in the new code file by appending it to the end of the code file.

If COPY_INPUT is FALSE, the new PED is either written on top of an existing PED within the original code file, or is appended to the end of the original code file.

During the construction of the program execution environment, each referenced library code file is checked for the presence of a nonzero checksum indicator in block zero of the segment dictionary information. If this checksum indicator is zero, the p-System checksum generation unit CHKSUMOPS is called to insert a valid checksum into the library code file. When the COPY_LIBRARIES parameter is TRUE, PED_GENERATE presents a prompt each time a library code file is updated with a new checksum.

This prompt asks if you wish the updated library code file to be copied to another file. When COPY_LIBRARIES is FALSE, no such prompts are displayed. A detailed description of the characteristics of this facility was given in the description of QUICKSTART, above.

The WRITE_PROGRESS_MESSAGES parameter is used to control whether or not PED_GENERATE writes progress messages and error messages to the file OUTPUT. These messages are generated when this parameter has the value TRUE and are suppressed when this parameter has the value FALSE. A detailed description of the format of the progress messages generated by PED_GENERATE was given in the description of QUICKSTART utility program.

The following is an example of the type of progress messages that would be written to the file OUTPUT as the result of a call to PED_GENERATE with the COPY_INPUT and WRITE_PROGRESS_MSGS parameters set to TRUE, and the COPY_LIBRARIES parameter set to FALSE:

```
Copying DISK1":PROG.code to DISK2:OLD.PROG.code
Copying complete.  (134 blocks copied)
Using KERNEL from *SYSTEM.PASCAL
Installing new checksum into *SYSTEM.LIBRARY
Using LONGOPS from *SYSTEM.LIBRARY
Including EXPR as segment of MYUNIT1 from DISK1:UNIT1.CODE
Using MYUNIT1 from DISK1:UNIT1.CODE
Using PASCALIO from *SYSTEM.PASCAL
```

The reference parameters THE_IORESULT and THE_NAME are used to return information pertaining to certain errors. Upon entry to PED_GENERATE, THE_IORESULT is set to the value zero, and THE_NAME is set to the empty string. Whenever PED_GENERATE returns the results PGR_INPUT_ERROR, PGR_OUTPUT_ERROR, or PGR_LIB_ERROR, the value of IORESULT is placed in THE_IORESULT. When PED_GENERATE returns the result PGR_UNIT_ERROR or PGR_LIB_ERROR the name of the unit or library file being referenced at the time of the error is placed into THE_NAME.

If the segment dictionary of the program indicates that the program must be linked using the p-System linker, PED_GENERATE halts and returns the result PGR_MUST_BE_LINKED_ERROR.

CHKSUMOPS Unit Interface

Here is the interface to the p-System CHKSUMOPS unit:

```
unit chksumops;

  interface

    const max_chksum_file_name_length = 255;

    type chksum_file_name = string[max_chksum_file_name_length];

         chksum_result =

           (chksum_no_error,
                    {Checksum operation
                     successful}

            chksum_obsolete_error,
                    {Checksum in code
                     file is obsolete;
                     that is, the contents
                     of the file have
                     been changed}

            chksum_io_error
                    {Error opening,
                     reading, or
                     writing code
                     file}
         );


    function chksum_gen(file_id: chksum_file_name;
                        var iorslt: integer):
                          chksum_result;

    function chksum_check(file_id: chksum_file_name;
                          var iorslt: integer):
                            chksum_result;
```

The CHKSUM_GEN function causes a new checksum to be calculated and installed in the checksum field of block zero of the code file specified by the FILE_ID parameter. This function returns the result CHKSUM_NO_ERROR if the operation is successful.

The CHKSUM_CHECK function calculates the correct checksum for the contents of the code file specified by the FILE_ID parameter. The calculated checksum is compared with the checksum stored in the file. If the checksum present in the code file isn't zero and doesn't match the calculated checksum, this function returns the result CHKSUM_OBSOLETE_ERROR; otherwise, the result CHKSUM_NO_ERROR is returned.

Both the CHKSUM_GEN function and the CHKSUM_CHECK function return the result CHKSUM_IO_ERROR whenever an I/O error is detected while opening, reading, or writing the specified code file. If this result is returned, the IORSLT parameter is set to the value of IORESULT to indicate the nature of the I/O error.

The checksum value zero is reserved to indicate the absence of a valid checksum in a code file.

CHAPTER 4

FILE MANAGEMENT UNITS

INTRODUCTION

Your Pascal programs can use the file management units to accomplish several tasks usually performed by the filer. There are four file management units:

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

DIR.INFO provides directory information. Your programs may use this unit to:

- List directories.
- Parse file names into volume ID, file name, file type, and size specification.
- Change file names.
- Change the date associated with a file or volume.
- Remove files.
- Krunch a volume.
- Mount and dismount subsidiary volumes.
- Grant exclusive access rights to a directory by task.
- Release those exclusive access rights.

WILD provides wild card string matching facilities.

FILE.INFO allows your programs to:

- Determine if files are opened.
- Find the length of a file.
- Determine what storage volume contains a given file.
- Extract the file title with its suffix, from a file.
- Find the starting block of a file.
- Determine whether or not a volume is a storage volume or a communications volume.
- Return the date associated with a file.

SYS.INFO allows your programs to:

- Determine the device number or volume name of the system disk (the volume referred to by asterisk, "*").
- Determine the file names for the work files and the volumes on which they reside.

INTERFACE SECTIONS

In order to take advantage of the file management units, your Pascal programs should use them in a USES declaration. (These units aren't available to FORTRAN and BASIC programs.) For example, to have access to all four units, you would use this declaration:

```
USES {$U wild.code} WILD,
     {$U dir.info.code} DIR_INFO,
     {$U sys.info.code} SYS_INFO,
     {$U file.info.code} FILE_INFO;
```

You can then call the routines these units contain from your programs. Here are the interface sections of the four file management units with embedded comments. The routines are described in detail throughout the rest of this chapter.

Unit Interface

```
Unit Wild;

Interface

Type

    D_PatRecP =                              D_PatRec;
    D_PatRec = Record
                 CompPos, { starting position of pattern in subject string }
                 CompLen, { length of pattern in subject string }
                 WildPos, { starting position of pattern in wild string }
                 WildLen : Integer; { length of pattern in wild card string }
                 Next : D_PatRecP;  { next pattern }
               End; { D_PatRec }

  Function D_Wild_Match(Wild, Comp : String; Var PPtr : D_PatRecP;
                        PInfo : Boolean) : Boolean;
  { Compares two strings (one containing wild cards) and returns true if they
    match.  Includes information about pattern matching that occurred if
    requested (by PInfo) }
```

Unit Interface

```
Unit Dir_Info;

Interface
  uses
    (*$U WILD.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;                 { Unit # of entry }
               D_Volume: String[7];             { volume name of unit }
               D_VPat: D_PatRecP;               { volume pattern info }
               D_NextEntry: D_ListP;            { Next entry in list }
               Case D_IsBlkd: Boolean Of
                 True: (D_Start,                { Starting block of entry }
                        D_Length: Integer;      { Length (in blocks) of entry }
                        Case D_Kind : D_NameType Of
                          D_Vol,                { Everything but D_Free }
                          D_Temp,
                          D_Code,
                          D_Text,
                          D_Data,
                          D_SVol: (D_Title: String[15];{ File name }
                                   D_FPat: D_PatRecP; { name pattern info }
                                   D_Date: D_DateRec; { File date }
                                   Case D_NameType of  { # of files on vol }
                                     D_Vol: (D_NumFiles: Integer)));
             End;

    D_Result = (D_Okay,          { Mission accomplished }
                D_Not_Found,     { Couldn't find name and/or type }
                D_Exists,        { Name already exists; no name change made }
                D_Name_Error,    { Illegal string passed }
                D_Off_Line,      { Volume not on line }
                D_Other);        { Miscellaneous error }

  Function D_Dir_List(D_Name: String; D_Select: D_Choice;
      Var D_Ptr: D_ListP; D_PInfo: Boolean): D_Result;
  { Creates pointer to list of names of specified NameTypes
```

```
  (D_Select), matching specified D_Name (wild card characters allowed).
  Includes information about pattern matching that occurred if requested
  (by D_PInfo) }

Function D_Scan_Title(D_Name: String; Var D_Volume, D_Title: String;
    Var D_Type: D_NameType; Var D_Segs: Integer): D_Result;
{ Parses D_Name }

Function D_Change_Name
    (D_OldName, D_NewName: String; D_RemOld: Boolean): D_Result;
{ Changes file name in D_OldName to name in D_NewName, removing already
  existing files of name in D_NewName if D_RemOld is set }

Function D_Change_Date(D_Name: String; D_NewDate: D_DateRec;
    D_Select: D_Choice): D_Result;
{ Changes date of directory or file name in D_Name to date specified by
  D_NewDate.  D_Name may contain wild cards }

Function D_Rem_Files (D_Name: String; D_Select: D_Choice): D_Result;
{ Removes file of specified name (wild cards allowed) }

Procedure D_Lock;
Procedure D_Release;
{ Provide means to limit use of DirInfo routines to one task at a time
  in multitasking environments }

Function D_Krunch (D_Unit, D_Block: Integer): D_Result;
{ Collects all unused space on a volume around D_Block. This unit must
  not be in use when this operation is performed.  }

Function D_Mount (D_File_Name : String) : D_Result;
Function D_DisMount (D_Vol_Name : String) : D_Result;
{ Provides a means of mounting and dismounting subsidiary volumes.
  Wild cards may be used.  }
```

Unit Interface

```
Unit Sys_Info;

Interface

  Type SI_Date_Rec = Packed Record
                       Month : 0..12;
                       Day : 0..31;
                       Year : 0..99;
                     End; { SI_Date_Rec }

  Procedure SI_Code_Vid (Var SI_Vol : String);
    { Returns name of volume containing current work file code }

  Procedure SI_Code_Tid (Var SI_Title : String);
    { Returns title of current work file code }

  Procedure SI_Text_Vid (Var SI_Vol : String);
    { Returns name of volume containing current work file text }

  Procedure SI_Text_Tid (Var SI_Title : String);
    { Returns title of current work file text }

  Function SI_Sys_Unit : Integer;
    { Returns number of bootload unit }

  Procedure SI_Get_Sys_Vol (Var SI_Vol : String);
    { Returns system volume name }

  Procedure SI_Get_Pref_Vol (Var SI_Vol : String);
    { Returns prefix volume name }

  Procedure SI_Set_Pref_Vol (SI_Vol : String);
    { Sets prefix volume name }

  Procedure SI_Get_Date (Var SI_Date : SI_Date_Rec);
    { Returns current system date }

  Procedure SI_Set_Date (Var SI_Date : SI_Date_Rec);
    { Sets current system date }
```

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; { F_Date_Rec }

  Function F_Open (var fid:  F_File_Type):boolean;

  (* returns true if the file is open and false if not open  *)

  Function F_Length (Var Fid : F_File_Type) : Integer;

    {Returns the length of the file attached to the Fid identifier.
     If the file is not opened result is returned as zero}

  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 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 there is no file opened to 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,
     the File_title is set to the 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.  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 storage device.  If there
     is no file opened for the Fid or if the unit is not a storage
```

```
    device, 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.}
```

DIRECTORY INFORMATION

This section describes the directory information unit, called DIR_INFO, which enables your programs to access file system information.

Many of your applications may need to access and modify directory information. This unit makes it easy to perform most of these sorts of operations. There are other ways to do this. The most common solution is to construct your own routines that directly access the operating system's data structures. However, the interfaces provided by this unit make directory information access much safer and easier.

The DIR_INFO unit provides the following capabilities to your programs:

- Directory Information Access. For any on-line storage volume, 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 mounting and dismounting subsidiary volumes.

- File Manipulation. DIR_INFO allows you to Krunch a volume in a similar fashion to the filer.

- 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.

- Multitasking 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.

Notation and Terminology

In this chapter, 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 < \ >. The following expression is an example:

```
<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. In this example, "fish" can be replaced by "trout," "salmon," or "tuna."

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 term <file-object> is used throughout this chapter; it is a generic term encompassing communications and storage volumes, files, and unused areas on storage volumes.

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. You should see the Operating System Reference Manual for a complete description of p-System files and file names if you aren't familiar with them.

Volume names and file names may contain wild cards (which are described in the next section). Device numbers and colons separating volume IDs and file names must appear literally; they must be independent of any wild card.

All DIR_INFO routines except D_Scan_Title ignore file length specification. In some cases, file name conventions in DIR_INFO differ slightly from p-System file name conventions:

- DIR_INFO considers an empty volume ID/file name argument to specify the prefix volume; that is, <file name> is empty (implying a volume reference), and <volume ID> is empty (implying the prefixed volume). An empty string isn't 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 p-System filer's interpretation of the (=) wild card. The filer doesn't accept file names of this form as volume specifiers.

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 storage volumes, temporary files, text files, code files, and other types of files.) The file type parameter is necessary because file names alone can't completely specify all types of file objects (such as unused disk areas). The routines that 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;
```

Here is a description of these scalar values:

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

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

- D_Temp. Selects all temporary files matching the file name argument. Files are considered temporary if they have been opened—and not yet closed—by a program.

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

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

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

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

File Dates

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

D_Date_Rec is declared as follows:

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

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

Error Results

All DIR_INFO routines that 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 isn't returned when routines return a result value indicating that an error has occurred.

The following items describe conditions that can cause errors:

- The specified files, volumes, or unused spaces can't 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.

An error can never cause a function to terminate abnormally. Errors that the routine can't identify explicity are flagged. This is done by returning a result that indicates an unknown error has occurred.

DIR_INFO defines the following 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);
```

You should refer to the descriptions of the various routines for details concerning the results of errors and the status of directory information returned during error conditions.

The DIR_INFO Routines

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

This function Krunches the files on the volume specified by D_Unit. This is similar to the filer's K(runch activity. The block indicated by D_Block is the point around which the unused disk space is consolidated. Files located before D_Block are moved forward (toward the directory) and files after it are 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 returns data to the previous—not the present—location of the file.

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

The D_File_Name parameter identifies an svol file. The corresponding subsidiary volume is mounted unless D_Result indicates otherwise. Wild cards may be used.

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

D_Scan_Title accepts the following parameters.

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

- D_VOLUME. A string that 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 uppercase characters and don't contain blank characters.

- D_TITLE. A string that returns the file name contained in D_NAME. If D_NAME doesn't contain a file name, D_TITLE is assigned the empty string. File titles assigned to D_TITLE contain only uppercase characters and don't contain blank characters.

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

The following items define 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 name in D_NAME is empty. D_TYPE is set to D_Code if the file name is terminated by ".CODE", or to D_Text if the file name is terminated by ".TEXT" or ".BACK". D_TYPE is set to D_SVOL if the file name 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 to determine what type it is. For example, the file name SYSTEM.COMPILER is returned as a data file because its suffix isn't .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.

Example Program

```
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 device numbers of storage and communications on-line volumes, number of files on storage volumes, 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.

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. To be listed in a directory, 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 D_List record fields return the following information for each file object in the D_Ptr list.

- D_Unit returns the device number of the device 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 section on the wild unit for details on pattern matching information). D_VPat is set to NIL if pattern matching information isn't 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 storage volume. 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 storage volumes (that is, 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 (that is 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 (such as 320 for a typical single density, 5-1/4" diskette).

- 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 (such as D_Kind in [D_Vol, D_Temp, D_Code, D_Text, D_Data, D_SVol]):

- D_Title contains the file name 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 information). D_FPat is set NIL if pattern matching information isn't 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 doesn't correspond to an .SVOL file. However, when accessed by its volume ID, the actual subsidiary volume returns 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 device that matches D_NAME (if D_Vol is in D_SELECT).

2. Files in directory of this volume that 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 device that matches D_NAME (if D_Vol is in D_SELECT).

5. Files in directory of this volume that 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. The D_WILD_MATCH function collects this information while comparing volume and file IDs; it is useful for determining how the wild cards were expanded in D_NAME. Information is returned in two pointers; one for volume names matched (named D_VPat) and one for file IDs matched (named D_FPat).

The following is an 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, CompLen is 5
   ('=' matches '.CODE')

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

NOTE: If the volume ID in D_NAME consists of a device number (such as "#5"), the volume assigned to the device 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 "MYDISK" resides in device 5.

1. WildPos is 1, WildLen is 2
   CompPos is 1, CompLen is 6
   ('#5' matches 'MYDISK')

NOTE: D_FPat and D_VPat never contain invalid information. If information is unavailable or hasn't 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_OffLine. Volume off-line. The volume specified by D_NAME wasn't on-line. This error occurs only when the volume ID in D_NAME doesn't contain wild cards (that is, a single volume is specified, and it is off-line). If the volume name in D_NAME contains wild cards but doesn't 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 couldn't 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*)
  Dirnfo;

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('Communications 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 { Listtest }
  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 doesn't 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 that accept wild cards.

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 a 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 off-line. The volume specified by D_OLD_NAME wasn't on-line. No change made.

- D_Other. Unknown. D_Change_Name encountered an error it couldn't identify. No change made.

Example Program

The following program demonstrates how you might use 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 Change

D_Change_Name doesn't 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 you have the following files:

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

You would like to change them to the following names:

```
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 concatenation 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.

Example Program

The following program demonstrates how you might use D_Change_Name and D_Dir_List when constructing a specialized file name changing utility. The program accepts a file name argument containing two '=' wild cards; for each file which matches the argument, the file title is changed by swapping the string patterns matched by the two '=' wild cards.

```
Program WildChange;

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 rebooted if the new date is to be used. 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.

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 isn't accepted by D_Change_Date in a new date.

- D_SELECT. A set of file and/or volume. All scalar types except D_Free and D_Temp apply to D_Change_Date. Disk free spaces identified by the D_Free scalar don't 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 are described in the following items.

- 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 off-line. The volume specified by D_NAME wasn't 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 doesn't 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.

Example Program

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]);
    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 wild cards. 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.

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. 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 can't 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 off-line. The volume specified by D_NAME wasn't 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 wild cards, but doesn't 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.

Example Program

```
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_Rem_Files 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_Rem_Files(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.

Example Program

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 }
```

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 that 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 Operating System Reference Manual that describes the filer operation. Because of the extra functions provided by this UNIT, there isn't a direct correspondence between the filer and this UNIT. Where there are differences in the use of characters, these are described.

Special Wild Card Characters

The following characters are defined as special characters:

question mark	?
equal 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"

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"

Equal Sign Wild Card

An equal 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"

Subrange Wild Card

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

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 doesn't 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 in the preceding paragraphs.

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      = an ASCII character
```

Examples of subrange wild card:

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

```
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) that describes how the strings were matched.

D_Wild_Match Parameters

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.

D_Wild_Match Pattern Matching Info

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 don't match, or the list wasn't requested (that is, 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 (that is, 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.

Example Program

The following program is an example of a string comparison routine that 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 : ');
```

SYSTEM INFORMATION

Unit SYS.INFO is 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 you require 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. Here are the SYS.INFO routines:

```
    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 }
```

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 Volume:

```
Function SI_Sys_Unit : Integer;
```

The SI_Sys_Unit function returns an integer function result. The device number of the drive 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 won't 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.

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 device ', 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 := 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}.
```

FILE INFORMATION

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 you to access the 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'.

```
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 be passed on as parameters to these procedures and functions.

```
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 isn't open. The following functions won't give the correct values if the file isn't 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 isn't opened, the result is returned as zero. This only has meaning for files on storage volumes 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 device number of the storage volume 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 device number (such as #4:). 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 storage volumes. 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 storage volume (or block-structured device). If there is no file opened for the Fid or if the device isn't a storage volume, 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;
```

CHAPTER 5

DEBUGGING AND ANALYSIS

INTRODUCTION

This chapter describes the debugger and the performance monitor. The debugger is a tool for correcting errors in programs that you develop. The performance monitor is a mechanism that may assist you in gathering program (or operating system) performance information.

DEBUGGER

The symbolic debugger is a tool for locating and correcting errors in compiled programs. You can call it from the Command menu. It can also be selected while a program is running (when a break point is encountered). Using the symbolic debugger, you may display and alter memory, single-step p-code, and display and traverse markstack chains.

To use the debugger effectively, you must be familiar with the p-machine architecture and understand the p-code operators, stack usage, variable and parameter allocation, and so on. These topics are discussed in the Internal Architecture Reference Manual.

You may have to use the Library utility to place the debugger into SYSTEM.PASCAL. If this is the case with your p-System package, you should consult the "Configuration Notes Appendix" to the Operating System Reference Manual.

Using the Debugger

There are no menus explaining the debugger commands because they would detract from any information displayed by the program being debugged. 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 entering the first character, press <space> to recall the main mode of the debugger.

A current compiled listing of the program is a helpful debugging tool. It helps you determine p-code offsets and similar information.

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

Entering and Exiting

Press 'D' to call the debugger from the Command menu. If you enter the debugger in a fresh state, the system displays the following prompts.

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

A fresh state means that the debugger wasn't previously active, and no break points are currently enabled. If you enter the debugger in a nonfresh state, only the left parenthesis "(" appears.

Exit the debugger by pressing 'Q' to select Q(uit, 'R' to select R(esume, or 'S' to select S(tep. The Q(uit option disables the debugger. If the debugger is selected again, it returns in a fresh state. The R(esume option won't disable the debugger and execution continues from where it left off. The debugger is still active; and if it is called again, it is in a nonfresh state. The S(tep option executes a single p-code and automatically again calls the debugger in a nonfresh state.

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

You may memlock or memswap the debugger (see the descriptions of those intrinsics) by using the M(emory command at the outer level. 'ML' memlocks and 'MS' memswaps the debugger.

Using Break Points

To enter the debugger while a program is running, but not alter the program's code, use the debugger to set break points. Press 'B' to call the B(reakpoint option and then use either the S(et, R(emove, or L(ist command. To set a break point, press S(et after pressing B(reakpoint. There are, at most, five break points 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 through 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 break point; and if that segment, procedure, and offset are encountered while resuming execution, the debugger is automatically called again.

Use a compiled listing of the program to determine the location of the break point. If no compiled listing is available, use the text file viewing facility.

To remove a break point, press B(reakpoint; then press R(emove. The system displays the following prompt:

```
Remove break #?
```

To remove a break point, enter its number; then press <space>.

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

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:

```
( L) 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 hexadecimal location 2C1A. Following this, eight bytes are displayed, first in HEXADECIMAL and then in ASCII (a dash "-" indicates that the character isn't 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 pressed, then the line may be altered in hexadecimal mode. If a '\' is pressed, then the line may be altered in ASCII mode. When altering in hexadecimal mode, any characters that are to be left unchanged may be skipped by pressing <space>. In the ASCII mode, any characters to be left unchanged may be skipped by pressing <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. Press 'C'. The U(p, D(own and L(ist options are available. If 'L' is pressed, 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.

Viewing Text Files

To view a text file from the debugger, press 'F' to call the F(ile command. 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 you wish 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 isn't available. Using this command, you can view source files on disk and disk files containing compiled listings without leaving the debugger.

Displaying Useful Information

Whenever control is returned to the debugger (that is, after a single-step operation, or when a break point 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 is displayed, the E(nable mode should be used. After entering '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 like 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 0#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(ddress and the system displays the following prompt.

```
Address ?
```

Enter the absolute address in hexadecimal. 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.

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 pressing <break>; control returns to the debugger.

Performance Monitor Interaction

The 'I' command calls the PM_Interactive procedure within the operating system if the performance monitor is enabled. You may, in this way, gain access to various sorts of program performance information while using the Debugger. For more information, see the "Performance Monitor" section later in this chapter.

The 'Z' Command

The 'Z' command displays the segment reference list for each segment which is currently associated (within the operating system as well as within the program being executed). This segment reference list is extracted from the environment vector (which is described in the Internal Architecture Reference Manual).

Segments within the currently executing program, and within the operating system, may contain external references; that is, they may call routines from another segment or access variables from another segment. For each segment which has external references, a list of the referenced segments is given by the 'Z' command. The names of the referenced segments as well as the associated segment numbers are given. (When two or more segments reference a particular segment, the number associated with the referenced segment may vary among them. This segment number is used in the p-code operators which call external routines and access external variables.)

For example, if you use the 'Z' command when the Debugger is called from the Command menu, the segment reference list for each unit within the operating system is displayed. Here is a partial listing:

```
the sib is GOTOXY
 1 KERNEL
 2 GOTOXY
 3 SCREENOP

the sib is DEBUGGER
 1 KERNEL
 2 DEBUGGER
 4 PDEBUGIN
 5 EXTRALEX
 6 EXTRAIO
 7 GOTOXY
 8 FILEOPS
 9 STRINGOP
10 PASCALIO
11 EXTRAHEA

the sib is PDEBUGIN
 1 KERNEL
 2 PDEBUGIN

the sib is SCREENOP
 1 KERNEL
 2 SCREENOP
 3 SEGSCINI
 4 STRINGOP
 5 SEGSCPRO
 6 PASCALIO
 7 EXTRAIO
 8 SEGSCCHE
 9 GOTOXY
```

The "sib" is the Segment Information Block (which is described in the Internal Architecture Reference Manual). After "the sib is," the name of an associated segment is given. Below this name are the segments it references along with the associated segment numbers.

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 break point at the beginning of the IF statement:

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

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

```
Hit break #0 at S=NOTDEBUG P#1 0#6
  (cd)  S=NOTDEBUG  P#1 0#6  SLD01
  (cd)  S=NOTDEBUG  P#1 0#7  SLDC1
  (cd)  S=NOTDEBUG  P#1 0#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' wasn't initialized. Looking over the listing, we quickly realize that this is the case.

Symbolic Debugging

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

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

To use symbolic debugging, it is necessary that the code being debugged is compiled with the $D+ option. The $D+ option, 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. Symbolic debugger information for a particular code segment is stored in another code segment. This other segment is given the same name as the code segment for which symbolic debugger information is generated. However, the name is in lowercase letters. (Executable code segments are always given names consisting of uppercase letters.)

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

```
Procname or #?
```

Enter the first eight characters of the procedure name. The next line displayed is:

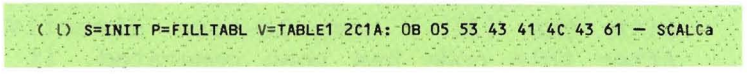
```
First#____ Last#____ Line#?
```

The underlines actually are values that define the range of line numbers available to you within the specified procedure. (These line numbers appear on compiled listings.) Enter the desired line number for the break point.

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 with $D+. The V(ar command allows specification of G(lobal, L(ocal, I(ntermediate, or P(rocedure variables in this manner. E(xtended variables aren't allowed to be specified symbolically. The V(ar command prompts:

```
Varname or Offset #?
```

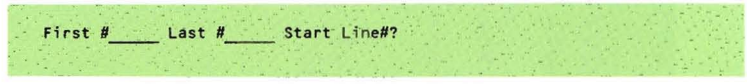
You may enter the first eight characters of the declared identifier. A line similar to the following appears:



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 requests:

Enter the first eight characters of the procedure name. The system displays the following prompt: End Line#?



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

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
(l ) S=EXAMPLE P=SETCIFD V=D  E7B2 : 0000 9448 BEE7 190C-H-
(Q )
```

The first time the debugger is entered, the program example isn't in memory and hence the symbolic segment isn't in memory. However, a break point can still be set symbolically providing you know on which line number to stop. For the second break point, 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 you try to access symbolically when the actual code segment is in memory and its symbolic segment counterpart isn't present, the system displays 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 (as described above) displays segment reference lists. For example, the following is a partial list for a program called EXAMPLE. The lowercase name 'example' is the segment produced by the compiler which contains the symbolic debugging information for 'EXAMPLE'. The existence of 'example' indicates that symbolic debugging information is available for at least one procedure in segment 'EXAMPLE'.

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

Summary of the Commands

A(ddress	Displays a given address.
B(reakpoint	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 from being displayed.
I(nteract	Interacts with the performance monitor.

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 performance 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, 'nonfresh'.
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(seg list	Displays segment lists.

PERFORMANCE MONITOR

You can gain access to performance information by writing a unit called PERFOPS and including it in the operating system. This performance information can help you analyze application programs or the p-System itself. In the future, a full implementation of PERFOPS will be provided. Currently, however, only the hooks for a performance monitor are available. You should be aware that a sophisticated understanding of the p-System's internal architecture is required in order to write a useful performance monitor.

The p-System expects the following interface for PERFOPS:

```
UNIT PERFOPS;
INTERFACE
  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. (PM_Start_Stop is responsible for setting this boolean.) 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 system calls this procedure from the fault handler. PM_Dump_Seg indicates that SegToDump is being removed from the code pool. (A SIB_P is a pointer to a SIB. SIBs are described in the Internal Architecture Reference Manual.) 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 should memlock PERFOPS and set Has_PM to true, or vice versa depending on the value of parameter start. You must call this routine before PERFOPS can be used. It should be called again to deactive PERFOPS. It can be called directly from the program being analyzed. It could also be called from a small program executed just prior to (and just after) the program being analyzed.

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

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

CHAPTER 6

UTILITY PROGRAMS

INTRODUCTION

The p-System's utilities are various precompiled programs that may assist you in many ways. Most of the utility programs included here are useful during program development. The utilities covered in this chapter are:

- The Decode utility which displays the content of code files in a meaningful fashion.
- The Native Code Generator which translates portions of a p-code file into processor-specific native code.
- The Patch utility which enables you to view the internal content of any sort of file.
- The XREF utility which is useful for analyzing Pascal programs.

DECODE

The DECODE utility, called DECODE.CODE, provides access, in symbolic form, to all useful items contained in code files. The following information is available.

- Names, types, global data size, and other general information about all code segments in the file.
- Interface section text, if present, for all units in the file.
- Symbolic listing of any (or all) p-code procedures in any (or all) segments of the file.
- Segment references and linker directives associated with code segments.

The decoder should be used whenever you want detailed knowledge of the internal contents of a code file; for instance, an implementor of a p-machine emulator decodes test programs so that the object code can be executed and understood step-by-step. You should refer to the Internal Architecture Reference Manual, if detailed use of the decoder is planned.

If a program uses a UNIT, the UNIT is decoded only if it is within the host file; DECODE won't search the disk for UNITs to decode. Assembly routines linked into a higher-level host won't be disassembled when the host is decoded.

When the system executes DECODE, the first prompt asks for the input code file (if necessary, the suffix .CODE is automatically appended). The next prompt asks for the name of a listing file to which DECODE's output may be written. This may be CONSOLE: (indicated by pressing <return>), REMOUT:, PRINTER:, or a disk file. The system then displays the following menu:

```
Segment Guide: A(ll), #(dct index), D(ictionary), Q(uit)
```

The following items explain the DECODE options.

D(ictionary	Displays the code file's segment dictionary.
A(ll	Disassembles all segments in the code file.
#(dct index	A number of a dictionary index followed by <return> disassembles a given segment, if present.
Q(uit	Exits the decoder.

DECODE Programming Example

Given the following Pascal program:

```
 1        0:d    1  {$L LIST1.TEXT}
 2        1:d    1  PROGRAM DEMO;
 3        1:d    1  VAR  I:INTEGER;
 4        1:d    2
 5        1:d    2  SEGMENT PROCEDURE ADDI;
 6   3    1:0    0  BEGIN
 7   3    1:1    0    I:=I+I;
 8   3    1:0    5  END;
 9   3    1:0    7
10   2    1:0    0  BEGIN
11   2    1:1    0    I:=50;
12   2    1:1    4    REPEAT
13   2    1:2    4      ADDI;
14   2    1:1    7    UNTIL I=400;
15         :0   14  END.
```

DECODE displays a prompt asking for input and listing file names. Then, if you press 'D' to call the D(ictionary option, the system displays the following listing.

```
 INX   NAME START SIZE VERSION M_TYPE SG# SEG_TYPE RL FMY_NAME or
                                                DSIZE  SGRF HISG TS
  0:   DEMO   2    20   IV_0  M_PSEUDO 2  PROG_SEG R     1     5    3  0
  1:   ADDI   1    14   IV_0  M_PSEUDO 3  PROC_SEG R    DEMO
  2:                                      NO_SEG
  3:                                      NO_SEG
  4:                                      NO_SEG
  5:                                      NO_SEG
  6:                                      NO_SEG
  7:                                      NO_SEG
  8:                                      NO_SEG
  9:                                      NO_SEG
 10:                                      NO_SEG
 11:                                      NO_SEG
 12:                                      NO_SEG
 13:                                      NO_SEG
 14:                                      NO_SEG
 15:                                      NO_SEG
(C):

Sex: LEAST significant byte first
                                  Next Page: 0
Segment Guide: A(ll, #(dct index), D(ictionary, Q(uit
```

The A(ll options produces the following disassembly.

```
Constant pool for segment DEMO
Block:   2  Block offset:   0  Seg offset:    0

   0:  1700 0000 4445 4D4F 2020 2020 0100 1400 0400 0000 ----DEMO    ----
  10:  0000                                               --

Block:   2  Block offset:  40  Seg offset:   40

   0:  0100 0000 0C00                                     ____

Segment: DEMO      Procedure:    1
Block:   2  Block offset:  26  Seg offset:   26
Data size:     0  Exist IC:   38
     Offset                                  Hex code
     0(000):   LDCB         50               8032
     2(002):   SRO           1               A501
     4(004):   SCXG    ADDI       1         7201
     6(006):   SLDO          1               30
     7(007):   LDCI        400               819001
    10(00A):   EFJ           4               D2F8
  exit code:
    12(00C):   RPU           0               9600

Constant pool for segment ADDI
Block:   1  Block offset:   0  Seg offset:    0

   0:  1300 0000 4144 4449 2020 2020 0100 1000 0400 0000 ----ADDI    ----
  10:  0000                                               --

Block:   1  Block offset:  32  Seg offset:   32

   0:  0100 0000 0C00                                     ____

Segment: ADDI      Procedure:    1
Block:   1  Block offset:  26  Seg offset:   26
Data size:     0  Exist IC:   30
     Offset                                  Hex code
     0(000):   SLDO          1               30
     1(001):   INCI                          ED
     2(002):   SRO           1               A501
  exit code:
     4(004):   RPU           0               9600
```

D(ictionary Display

DECODE's D(ictionary option display is a format of the code file segment dictionary. The following items describe the information that is displayed.

Index — DECODEs name for each segment; individual segments may be disassembled by entering their number and pressing <return>; for example, '0<return>' for this sample causes only DEMO to be disassembled.

Name — Contains the names of each segment.

Start — Contains each segment's starting block (relative within the code file).

Size — The length in words of each segment.

Version — The p-System version number of the segment.

M_TYPE is the machine type. Usually this is M_PSEUDO, indicating a p-code segment, but assembled segments indicate a given machine. Other possible values for M_TYPE are M_6809, M_PDP, M_8080, M_Z_80, M_GA_440, M_6502, M_6800, M_9900, M_8086, and M_68000.

SEG_TYPE can be NO_SEG, PROG_SEG, UNIT_SEG, PROC_SEG, or SEPRT_SEG. NO_SEG is an empty segment slot, PROG_SEG is a program segment, UNIT_SEG is a UNIT segment, PROC_SEG is a SEPARATE routine segment, and SEPRT_SEG is an assembled segment.

The RL columns indicate whether or not the segment is relocatable and whether it needs to be linked. An 'R' indicates a relocatable segment. An 'L' indicates a segment that must be linked.

If the segment is declared within a program or unit, then the FMY_NAME column contains its family name, that is, the name of the program or unit. Otherwise, the DSIZE SGRF HSG columns are displayed and contain, respectively, the compilation module's data size, segment references, and the maximum number of segments.

At the bottom of the screen, '(C):' is followed by whatever copyright notice the code file may have. The next line indicates the byte sex of the code file. The menu is the last line on the screen. On the same line, the block number of the next portion of segment dictionary is displayed after "Next Page:". (In this example, the segment dictionary is entirely contained in block zero so next page is zero. The last portion of the segment dictionary always points back to block zero.)

Disassembled Listing

The first portion of a disassembled listing shows the housekeeping information at the beginning of a code segment. The block number of this information is given. (Code files start at block 0.) The block offset and segment offset are always 0. The information occupies the first 11 words (0 through 10) of the segment. This housekeeping information (which is described in the Internal Architecture Reference Manual) includes such things as the segment name, byte sex indicator word, part number, and so forth. To the right, the same information is displayed as ASCII characters when printable, and as dashes when nonprintable. (The segment name is usually the most obvious part of this display.)

The next few lines have the same format and display the constant pool. The block offset and segment offset are always nonzero for the constant pool. They represent the offset, in bytes, of the constant pool from the beginning of the block and the beginning of the segment, respectively. String constants and character type constants are usually easy to pick out in the ASCII display to the right.

The disassembled code itself is displayed by procedures. The block number, block offset, segment offset, data size, and Exit IC are displayed. (Data size and Exit IC are described in the Internal Architecture Reference Manual.) The OFFSET column shows the offset in bytes from the front of the procedure (the count is in both decimal and hexadecimal). Then the p-code mnemonic is displayed; followed by the operands, if any; and finally, the HEX CODE for that particular instruction.

The OFFSET column corresponds to the fourth column in a compiled listing.

Jump operands are displayed as offsets relative to the start of the procedure, rather than IPC-relative (IPC is the instruction program counter). This is to make the disassembly more readable. Thus, the operand shown is the offset of some line; in the example, the equal false jump (EFJ) on line 10 shows 4, which means line 4—the SCXG instruction; the HEX CODE indicates that the offset is actually F8 (or -8), which is IPC-relative.

If a single segment were to be disassembled (rather than using the A(ll) command), a line similar to the following would be displayed.

```
There are 1 procedures in segment DEMO.
Procedure Guide: A(ll), #(of procedure), L(inker info),
                 C(onstant pool), S(egment references),
                 I(nterface text), Q(uit)
```

Selecting A(ll) disassembles all of the procedures in the segment (in the example there is only one). Entering the number of a procedure followed by <return> disassembles that procedure. If present, L(inker information, S(egment references, and I(nterface text may also be displayed.

For example, if the segment is a unit with interface text and you press 'I', the following listing may be displayed.

```
Interface text for segment SOMEUNIT:

PROCEDURE A_PROC;
PROCEDURE ANOTHER_PROC(I:INTEGER);
FUNCTION A_FUNCTION:BOOLEAN;
IMPLEMENTATION
```

If the segment had references to other segments and you press 'S', the following listing may be displayed.

```
Segment references list for segment KERNEL:

14: ***             5: SYSCMND
13: CONCURRE        4: DEBUGGER
12: PASCALIO        3: FILEOPS
11: HEAPOPS         2: SCREENOP
10: STRINGOP        0:
```

If the segment had linker information and you press 'L', the following listing may be displayed.

```
Linker information for segment SOMESEG:

SOMEPROC EXTPROC srcproc=4 nparams=0 koolbit=false
```

NATIVE CODE GENERATOR

The Native Code Generator (NCG) is a utility program that translates selected portions of an executable p-code file into processor-specific native code (n-code). Using native code directives inserted into the source code, you indicate 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 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.

You indicate what 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-

You insert the first switch §$N+} where the translation should begin and insert the last switch {$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 UCSD Pascal. Similar notations apply to other languages.)

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 can't 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.

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. You can specify translation on a statement-by-statement basis. The following example shows the use of native code directives in BASIC.

```
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
```

Directives and FORTRAN

To designate code for translation in a FORTRAN source code file, you 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
```

Running the NCG

The NCG is run by executing the appropriate code file (such as Z80.NCG.CODE).

The NCG generates a prompt asking you 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 you 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 <return> 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 de
          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 emulator (PME) is achieved using the following conventions:

- NATIVE is the p-code that instructs the PME 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 processor 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.

NCG LIMITS

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) are treated by the p-machine as a single p-code. (See individual processor listings.) This fact causes two problems. First, although the <break> key may be recognized by the p-machine emulator (PME) at any point, no further action is taken until the next p-code boundary (that is, 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 can't 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. You may force a p-code procedure call by calling an empty procedure. P-code operators which perform procedure calls aren't translated into n-code. Therefore, 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 won't translate. For example, using the Pascal primitive, P_Machine, to generate an RPU instruction.

PATCH

The Patch utility enables you to view files and alter them interactivley on the byte level.

Patch is meant to be used interactively with a CRT. It uses the screen control module (see the Internal Architecture Reference Manual) to accomplish this; therefore, it is terminal-independent (within limitations).

There are two main facilities in Patch: a mode for editing files on the byte level and a mode for dumping files in various formats.

The byte-editing capability allows you to edit text files, make quick fixes to code files, and create specialized test data.

The dump capability provides formatted dumps in various radices. It also allows dumps from main memory.

EDIT Mode

When the system executes Patch, you are in the EDIT mode. DUMP is reached by entering 'D'. No information is lost in chaining back and forth between the two modes.

EDIT allows you to open a file or device, read selected blocks (specified by relative block number) into an edit buffer, either view that buffer or modify it (with TYPE), and write the modified block back to the file. The system displays buffers on the screen in the desired format; these can be edited in a manner similar to using the screen-oriented editor.

The following paragraphs describe the individual commands of the EDIT mode. When it is impossible to perform a command, Patch responds with self-explanatory error messages. The following lines are the EDIT mode menu.

```
EDIT : D(ump, G(et, R(ead, S(ave, M(ix, T(ype,I(nfo, F(or, B(ack, ?
EDIT : V(iew, W(ipe, Q(uit, ?
```

The following items explain each menu option.

D(ump	Calls DUMP.
G(et	Opens the file or device and reads block zero into the buffer.
R(ead	Reads a specified block from the current file.
S(ave	Writes the contents of the buffer out to the current block.

M(ix	Changes the display format for the current block. Pressing 'M' toggles to change from one format to another: hexadecimal or mixed.
Mixed	Displays printable ASCII characters and the hexadecimal equivalent of nonprintable characters.
Hex	Displays the block in hexadecimal digits.
I(nformation	Displays information about the current file including the file name, the file length, the number of the current block, whether the file is open, whether UNITREADs are allowed, the device number (-1 if UNITIO is false), and the byte sex of the current machine.
F(orward	Gets the next block in the file.
B(ackward	Gets the preceding block in the file.
V(iew	Displays the current block (see M(ix).

W(ipe	Clears the display of the block off the screen.
Q(uit	Quits the Patch program.
T(ype	Goes into the typing mode, which allows the buffer to be edited (described in following section).

TYPE Mode

The TYPE mode, like the screen-oriented editor, allows the information on the screen to be modified by moving the cursor and entering over existing information. To correct errors made while using the TYPE mode, leave the EDIT mode without saving the file, read the block over, and try again.

The following line is an example of the TYPE mode menu.

```
TYPE: C(har, H(ex, F(ill, U(p, D(own, L(eft, R(ight, <vector arrows>, Q(uit
```

C(haracter	Exchanges bytes in the buffer for ASCII characters as they are pressed, starting from the cursor and continuing until you press <etx>. Only printable characters are accepted.

H(ex — Exchanges bytes in the buffer for hexadecimal digits as they are pressed, starting from the cursor and continuing until a 'Q' is pressed; (hexadecimal digits can be either uppercase or lowercase).

F(ill — Fills a portion of the current block with the same byte pattern. Accepts either ASCII characters or hexadecimal digits for the pattern; upon completion, the cursor rests after the last byte filled.

The following commands move the cursor around within the block of displayed data. The cursor is always at a particular byte. Rather than moving off the screen, the cursor wraps around from side to side and from top to bottom.

U(p — Moves the cursor up one row.

D(own — Moves the cursor down one row.

L(eft — Moves the cursor left one column.

R(ight — Moves the cursor right one column.

<vector arrows>	Moves the cursor in the direction of the arrow.
Q(uit	Exits the TYPE mode and returns to the EDIT mode.

DUMP Mode

You can generate DUMP mode in the following formats:

- Decimal, hexadecimal, and octal words.
- ASCII characters, if printable.
- Decimal (BCD) and octal bytes.

DUMP can flip the bytes in a word before displaying it or simultaneously display a line of words in both flipped and nonflipped form.

Input to the DUMP mode can be a disk file you specify or can come directly from main memory. (The DUMP mode is used primarily to examine the PME and/or the Basic Input/Output Subsystem [BIOS].)

The width of the output can be controlled; a line may contain any number of machine words: 15 words fill an 132-character line, and 9 words fill an 80-character line.

When you enter the DUMP mode, the screen displays two options: D(o and Q(uit. Also a lengthy set of format specifications are displayed. These can be modified by pressing the letter of the item and then entering the specification. To activate the specification, press 'D' for D(o.

The following list shows the DUMP mode specifications:

a. The input: A disk file or device.

b. The number of the block from which dumping starts; if (A) is a device, this number isn't range-checked.

c. The number of blocks to print out; if this is too large, DUMP merely stops when there are no more blocks to output.

d. Pressing 'D' starts the dump.

e. A toggle: If true, it reads from main memory; if false, it reads from the file in (A).

f. An offset: The dump may start with a byte that is past byte zero; 0 <= (F) <= maxint.

g. The number of bytes to print; 0 <= (G) <= maxint.

h. The output file, opened as a text file.

i. The width of the output line, in machine words; 1 <= (I) <= 15.

The following six items have three associated Booleans that must be specified: USE, FLIP, and BOTH.

USE tells DUMP whether or not to use the format associated with that item.

FLIP tells DUMP whether or not to flip the bytes before displaying words in that format.

BOTH tells DUMP to simultaneously display both flipped and nonflipped versions of the line. If BOTH is true, the value of FLIP doesn't matter.

j. Display each word as a decimal integer.

k. Display each word as hexadecimal digits in byte order.

l. Display each word as ASCII characters in byte order; nonprintable characters are displayed as hexadecimal digits.

m. Display each word as an octal integer; this is the octal equivalent of (J).

n. Display each word as decimal bytes (BCD) in byte order.

o. Display each word as octal digits in byte order.

s. Put a blank line after the nonflipped version of a line.

t. Put blank lines between different formats of a line.

Both the EDIT and DUMP modes remember all their pertinent information when the other mode is operating.

Prompts

All user-supplied numbers used by PATCH are read as strings and then converted to integers. Only the first five characters of the string are considered. If there are any nonnumeric characters in the string, the integer defaults to zero. If integer overflow occurs, the integer defaults to maxint. (Since integer overflow can only be detected by the presence of a negative number, integers in the range 65536 to 98303 come out modulo 32768.)

THE XREF UTILITY
THE CROSS-REFERENCER

Introduction

The procedural cross-referencer (XREF) is a software tool that helps you interpret large Pascal program listings. The referencer provides a compact summary of the procedure nesting in a program; a list of the procedures; and, for each, the procedures that call them; and a table of calls each procedure made along with all nonlocal variable references. It thus provides information about the interprocedural dependencies of a program.

Referencer's Output

The referencer produces five tables and an optional warnings file:

- Lexical structure table: summarizes static procedure nesting.
- Call structure table: lists procedures and the procedures that they call.
- Procedure call table: presents procedures and the procedures that call them.
- Variable reference table: shows each procedure and the variables it references.

- Variable call table: lists each variable and the procedures which reference or modify it.

- Warnings file if desired: indicates possible problems in the source program.

Lexical Structure Table

The first table displays the lexical structure and the procedure headings. (The term procedure means procedure, function, process or program in this document unless otherwise stated.) As the system reads the input program, it prints out each heading with the line numbers of the lines in which it occurs. The text is indented to display the lexical nesting. (This indentation must sometimes be compressed to fit on an output line.)

Referencer considers a procedure heading to be any text between the words: procedure, function, process, or program—and the semicolon which follows. This isn't the Pascal definition, but is more useful in debugging programs. If these reserved words are embedded within comments, they are ignored.

The Call Structure Table

The system produces the second table after it scans the program completely. The call structure table is the result of examining the internal data. For each procedure listed in alphabetical order, the table holds:

- The line-number of the line on which its heading starts.
- Unless it was external or formal (and had no corresponding block), the line number of the BEGIN that starts its statement part.
- The characters 'ext' if the procedure has an external body (declared with a directive other than FORWARD); the characters 'fml' if it is a formal procedural or functional parameter; or 'eh?' if it is declared forward with no associated forward block or BEGIN. If a number appears, the procedure has been declared FORWARD and this is the line number of the line where the block of the procedure begins (that is, the second part of the two-part declaration).

- A list of all user-declared procedures directly called by this procedure. (In other words, their call is contained in the statement part.) The list is in order of occurrence in the text; a procedure isn't listed more than once.

- A list of variables referenced by this procedure; and, if nonlocal, the procedure in which they were declared. If a variable is modified by an assignment, then it is printed with an asterisk (*) in front of it.

The Procedure Call Table

This table is an alphabetical list of procedures; and for each procedure the procedures that call it.

Variable Reference Table

This table is an alphabetical list of procedures; and, for each procedure, the variables that the procedure examines or modifies in any way. If the variable isn't local to the procedure in question, then the procedure is listed in which the variable was declared.

Variable references are shown in three forms:

- <variable name> ::= a local variable
- <procedure name> <variable name> ::= a variable defined in <procedure> that is used but not modified
- <procedure name>*<variable name> ::= a variable defined in <procedure> which is modified

Variable Call Table

The form of the variable call table is demonstrated in the following line.

```
<procedure name> <variable name>: <procedure name> [<procedure name>]
```

The first procedure name is the procedure that owns the variable name, and the following procedure(s) either examine or modify that variable.

Warnings File

A file of warning messages. There are three types of warning messages in the warning file:

- 'Symbol' may be undeclared line# xxxx.
- 'Symbol' may not be initialized line# xxxx.
- Not standard, nested comments line# xxxx.

'Symbol' is an identifier, and xxxx is the number of the line on which it occurs.

Referencer only catches initializations done by replacement statements (':='), so variables that are initialized by procedure calls (including READ, and so on) are flagged as possibly uninitialized. Depending on the program, there may be a surplus of such warning messages.

The 'Not standard, nested comments' warning refers to the nesting of comments having different bracket types: (* like this { verstehen Sie? } *), which is accepted by the UCSD Pascal compiler, but not the current ISO draft standard.

The warnings file may only be generated if the variable reference table is also generated.

Using Referencer

The referencer has options that are user-defined at run-time. When you X(ecute XREF, referencer displays prompts asking for answers for the following questions.

- Width of the output device? [40..132]

  This is the length of the output line for the available terminal/printer. Suggested output width is 80 characters.

- File to be Cross-Referenced?

  The name of the text file that contains the Pascal program to be referenced. If the specified file can't be successfully opened, the prompt is repeated until you enter a valid input file name or press <return>. Entering an empty file name, (<return>) exits referencer.

- Is this a compiled listing? [y/n]:

  The program reads either .TEXT files containing Pascal source programs or listing files generated by the compiler. Using a compiled listing as input assures you that the line numbers referenced are synchronized with the line numbers the compiler generates.

- Do you want intrinsics listed?

  This allows identifiers such as 'WRITELN', 'PRED', and 'GET' to be accepted as valid symbols. These are then cross-referenced as procedures listed outside the lexical nesting and, therefore, aren't expected to have a 'BEGIN' associated with them.

- Do you want initial procedure nestings?

  This generates the lexical structure table. This table shows the procedure headings and, for each procedure, the list of procedures that it calls.

- Do you want procedure called by trees?

  This option is offered only if the lexical structure table is desired. A 'y' generates both the call structure table and the procedure call table. The procedure call table lists each procedure and all of the procedures that call it. (A warning is displayed if less than 10,000 words of memory are available to generate these trees; no provision is made for possible stack overflow.)

- Do you want variables referenced? [y/n]:

  A 'y' generates the variable reference table.

- Do you want variable called by trees? [y/n]:

  A 'y' generates the variable call table.

- Do you wish warnings? [y/n]:

  'Y' generates the warnings file. This option is offered only if the preceding selection was made.

- Please enter the name of the warning file:

  If you select warnings, then you have the option of directing the warnings to any file. If the file is a disk file, the name should have '.TEXT' appended to it.

- Output File:

  The name of the file to which you would like the output directed. If the file is a disk file, the name should have '.TEXT' appended to it.

The referencer expects to read a complete and syntactically correct Pascal program. Although results with syntactically incorrect programs aren't assured, the referencer isn't sensitive to most flaws. It cares about procedure, function, program headings, and about properly matching BEGINs and CASEs with ENDs in the statement parts.

Referencer doesn't try to format procedure and function headings; it leaves them as they were entered in the program, except for aligning indentations.

The tables are all as wide as the output line length, as specified by you. Eighty characters are usually sufficient. For large programs, the first table (the lexical structure table) is clearer with a larger print line.

Limitations

When presented with incorrect Pascal programs, the behavior of referencer isn't assured. However, it has been designed to be reliable, and there are few flaws that can cause it to fail. The most critical features are: (1) the general structure of procedure headings; and (2) correctly matching an END with each BEGIN or CASE in each statement part (since this information is used to detect the end of a procedure).

If an error is explicitly detected (referencer has very few explicit error checks and minimal error-recovery), the system displays the following message:

```
FATAL ERROR - No identifier after prog/proc/func - At Line No. ###
```

The line number displayed (###) is the line where the program found an error; like all diagnoses this doesn't assure that the correct reason is ascribed to the error. Processing continues for a while despite the fatal error, but only the lexical structure table is produced.

Referencer accepts standard Pascal programs, UCSD Pascal programs, and p-System units; it processes each correctly.

APPENDICES

APPENDIX A
EXECUTION ERRORS

0	Fatal system error
1	Invalid index, value out of range
2	No segment, bad code file
3	Procedure not present at exit time
4	Stack overflow
5	Integer overflow
6	Divide by zero
7	Invalid memory reference <bus timed out>
8	User break
9	Fatal system I/O error
10	User I/O error
11	Unimplemented instruction
12	Floating point math error
13	String too long
14	Halt, Break Point
15	Bad Block
16	Break Point
17	Incompatible Real Number Size
18	Set Too Large
19	Segment Too Large

All run-time errors cause the system to I(nitialize itself; FATAL errors cause the system to rebootstrap. Some FATAL errors leave the system in an irreparable state, in which case you must rebootstrap.

APPENDIX B
I/O RESULTS

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Device Number
3	Illegal I/O request
4	Data-com timeout
5	Volume is no longer on-line
6	File is no longer in directory
7	Bad file name
8	No room, insufficient space on volume
9	No such volume on-line
10	No such file on volume
11	Duplicate directory entry
12	Not closed: attempt to open an open file
13	Not open: attempt to access a closed file
14	Bad format: error in reading real or integer
15	Ring buffer overflow
16	Volume is write-protected
17	Illegal block number
18	Illegal buffer

APPENDIX C
DEVICE NUMBERS

Device Number	Volume Name
1	CONSOLE:
2	SYSTERM:
4	<System disk '*'>
5	<other disk>
6	PRINTER:
7	REMIN:
8	REMOUT:
9...127	<additional disks, subsidiary volumes, or user-defined serial devices>
128...255	<user-defined devices>

APPENDIX D
ASCII TABLE

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	78	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	89	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

APPENDIX E
PASCAL SYNTAX ERRORS

1: Error in simple type
2: Identifier expected
3: Unimplemented error
4: ')' expected
5: ':' expected
6: Illegal symbol (terminator expected)
7: Error in parameter list
8: 'OF' expected
9: '(' expected
10: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: Error in <field-list>
20: '.' expected
21: '*' expected
22: 'INTERFACE' expected
23: 'IMPLEMENTATION' expected
24: 'UNIT' expected

50: Error in constant
51: ':=' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in <factor> (bad expression)
59: Error in variable

60: Must be of type 'SEMAPHORE'
61: Must be of type 'PROCESSID'
62: Process not allowed at this nesting level
63: Only main task may start processes

101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of the appropriate class
104: Undeclared identifier
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure
parameter
117: Unsatisified forward reference
118: Forward reference type identifier in
variable declaration
119: Respecified parameters not OK for a
forward declared procedure
120: Function result type must be scalar,
subrange or pointer
121: File value parameter not allowed
122: A forward declared function's result type
cannot be respecified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure
parameter

126: Number of parameters does not agree
with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be Boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the
declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting
expression
148: Subrange bounds must be scalar
149: Index type must be integer

150: Assignment to standard function is not
allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or nonlocal
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed

160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not
allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Value parameter expected
172: Undeclared external file
173: FORTRAN procedure or function expected
174: Pascal functon or procedure expected
175: Semaphore value parameter not allowed
176: Undefined forward procedure or function

182: Nested UNITs not allowed
183: External declaration not allowed at this
nesting level
184: External declaration not allowed in
INTERFACE section
185: Segment declaration not allowed in
INTERFACE section
186: Labels not allowed in INTERFACE section
187: Attempt to open library unsuccessful
188: UNIT not declared in previous uses
declaration
189: 'USES' not allowed at this nesting level
190: UNIT not in library
191: Forward declaration was not segment
192: Forward declaration was segment
193: Not enough room for this operation
194: Flag must be declared at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure
entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
398: Implementation restriction
399: Implementation restriction

400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: 'PROGRAM' or 'UNIT' expected
406: Include file not legal
407: Include file nesting limit exceeded
408: INTERFACE section not contained in one
file
409: Unit name reserved for system
410: Disk error

500: Assembler error

APPENDIX F
COMPILER BACK-END ERRORS

The 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 (such as bad blocks on the disk), or source file information that is skipped by the front-end but used by the back-end.

The following table lists each of the back-end errors and gives a possible explanation for their occurrence:

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 isn't in the scope of the compilation unit.

7 The back-end can't 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't be found in the jump table.

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't 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 code pointer 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 Couldn't find a particular segment in the intermediate code file.

99 The number of procedures doesn't match the number specified in the procedure dictionary.

When you encounter a back-end error:

- 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.

INDEX

- 8 -

- A -

- B -

- C -

- D -

- E -

- F -

Function

- G -

- H -

- I -

- K -

-L-

-M-

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Y -

- Z -