UCSD p-System™ Program Development

Software Library Manual

Texas Instruments Professional Computer

Cover Part No. 2232346-0001

UCSD p-System Program Development
2232399-0001
Original Issue: 15 April 1983

Preface

This book is a reference manual for the UCSD p-System™* on the Texas Instruments Professional Computer. It describes the p-System facilities which enable you to develop programs. It is designed for a processing professional who is familiar with the p-System. The UCSD Pascal™* programming language is not described in this manual.

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

UCSD Pascal Reference Manual
UCSD p-System 8086 Assembler Reference Manual
UCSD p-System Internal Architecture Reference Manual
UCSD p-System Optional Products Reference Manual
UCSD p-System FORTRAN-77 Reference Manual
UCSD p-System BASIC Reference Manual
UCSD p-System Assemblers Reference Manual
UCSD p-System 8086/88/87 Assembler Reference Manual
UCSD p-System Adaptable System Installation Manual

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 nor liable for any consequences of their use.

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

Contents

Preface iii

1 Introduction 1-1
How to Use This Manual 1-1
Background 1-2
Design Philosophy 1-3

2 Compiling Programs and Units 2-1
Introduction 2-3
Using the Compiler 2-3
Segments, Units, and Libraries 2-27
General Tactics 2-32

3 User Interface 3-1
Introduction 3-3
Run-Time Application Facilities 3-3
The Screen Control Unit 3-6
Error Handler Unit 3-14
The Command I/O Unit 3-17
Simple Color Interface 3-20
Turtlegraphics 3-22

4 File Management Units 4-1
Introduction 4-3
Interface Sections 4-4
Directory Information 4-10
Wild Cards (WILD) 4-49
System Information (SYS.INFO) 4-57
File Information (FILE.INFO) 4-61
Time Date Unit 4-63

5 Debugging and Analysis 5-1
Introduction 5-3
Using the Debugger 5-3
Symbolic Debugging 5-12
Summary of the Commands 5-17

6 Utilities 6-1
Introduction 6-3
Decode 6-3
The Library Utility 6-12
Native Code Generator 6-17
Patch 6-21
Print Spooling 6-28
REALCONV Utility 6-29
XREF — the Cross-Referencer 6-31

Appendixes

A Special Keys

B Execution Errors

C I/O Results

D Device Number Assignments

E ASCII Codes

F Keyboard Mapping

G Pascal Compiler Syntax Errors

H Pascal Compiler Back-end Errors

1

Introduction

HOW TO USE THIS MANUAL

This book is a reference manual for use with the UCSD p-System on the Texas Instruments Professional Computer. It describes the p-System facilities which enable you to develop programs.

Chapter 2, Compiling Programs and Units, covers the Pascal compiler. The UCSD Pascal programming language is not covered in this manual. You should see the *UCSD Pascal* reference manual, TI Part Number 2232401-0001, 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 Interface, 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 can 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 Units, 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 program debugging and analysis aids.

Chapter 6, Utilities Programs, describes several utility programs that are generally useful. Most of them are specifically used 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 machinespincluding the Z80™[1], 8080/8085, 8086, 6502, 6809, 68000, 9900, PDP-11™[2], LSI-11™[2], and VAX™[2].

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 one thousand inquiries.

[1] Trademark of Zilog Incorporated.

[2] Trademark of Digital Equipment Incorporated.

As interest grew, the demand for the p-System could not 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.

Now the p-System is available on the Texas Instruments Professional Computer.

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:

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

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

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

The current design philosophy at SofTech Microsystems, where the p-System continues to evolve, is basically 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 can 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 UCSD p-System.

2

Compiling Programs and Units

Introduction 2-3
Using the Compiler 2-3
Syntax Errors 2-6
Compiled Listings 2-7
Compiler Options 2-11
$B — Begin Conditional Compilation 2-12
$C — Copyright Field 2-13
$D — Conditional Compilation Flag 2-13
$D — Symbolic Debugger 2-13
$E — End Conditional Compilation 2-13
$I — I/O Check Option 2-13
$I — Include File 2-14
$L — Compiled Listing 2-15
$N — Native Code Generation 2-16
$P — Page and Pagination 2-16
$Q — Quiet 2-17
$R — Range Checking 2-17
$R — Real Size Directive 2-17
$T — Title 2-18
$U — Use Library 2-18
$U — User Program 2-19
Conditional Compilation 2-19
Selective Uses 2-22
Segments, Units, and Libraries 2-27
Segmenting a Program 2-27
Separate Compilation — Units 2-28
Libraries 2-29
General Tactics 2-32

INTRODUCTION

This chapter is principally concerned with the UCSD Pascal compiler on the Texas Instruments Professional Computer. The UCSD Pascal programming language is not covered here. If you are interested in a detailed description of UCSD Pascal, consult the *UCSD Pascal* reference manual.

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 8086 assembly language and runs directly on the Texas Instruments Professional 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 do not 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. Do not include the *.CODE* suffix (which, once again, is assumed). If you simply press the **RETURN** key, the code file *SYSTEM.WRK.CODE is produced. The next prompt is:

```
Output file for compiled listing ? (<cr> 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 by pressing the **RETURN** key. When you enter a file name, the listing is placed in the file. You can use the suffix *.TEXT*, but it is always appended if you do not. If you specify a communications volume, the listing is sent there (where it is printed, displayed, or transmitted). When you simply press the **RETURN** key, no listing is produced. If you wish to exit the compiler, rather than respond to either the code file name prompt or the listing destination prompt, press the **ESC** key followed by the **RETURN** key.

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 overridden. (You should note that *.TEXT* is not automatically appended with the $L option 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 the **RETURN** key in response to the compiler's `To what code file?` prompt. 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 does not 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 is not present, you need to look up the error number in the appropriate appendix to this manual.

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

You can press the space bar 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 the ESC key). However, you can now fix the error immediately because the editor is automatically called. If the file that you are compiling is a work file, it is read into your work space. If it is not 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 the space bar 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 menu for a listing file which is described above. Alternatively, you can use the $L compiler option which is described in the following section entitled compiler options.

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

```
Pascal Compiler VERSION      1/ 1/83        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 are not 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:

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

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

- 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.

- 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 the letter *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 the letter *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 sections from the text that you are specifically compiling.

Here is a portion of a compiled listing that 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(Page Inx< lastpage item,'bad Page Inx');
---> 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 can 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 cannot contain an asterisk (*).

Some options can 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 plus (+) turns it on, and a minus (−) turns it off. The options I and R are stack options, as are the conditional compilation flags.

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 caret ^ (thus reenabling 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 negative (−).

```
{$I−} ... current value is '− − no I/O checking
      ...
{$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 do not 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 display unit).

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 following section on conditional compilation.

$D — Symbolic Debugger

D is a toggle option. $D+ turns on the symbolic debugging information. $D−, which is the default, turns this off. For more information see Chapter 5 on the Debugger.

$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 was not successful, 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 do not 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, the end user may enter something incorrect from the keyboard. Rather than terminating with an I/O error, your program may prompt the user 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 can 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 can be nested to a maximum of three files deep.

Note that if a file name begins with a plus (+) or a minus (−), 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 do not specifically name a listing file and $L+ is in effect, the compiler writes to *SYSTEM.LST.TEXT.

You should note that listing files that are sent to disk files can 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 toggle switch option. The $N+ tells the compiler to output information to start native code generation. The $N−, which is the default, tells the compiler to end code generation.

$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 (American National Standard Code for Information Interchange Form Feed) is output every 66 lines if $P+ is in effect (this is the default). If you do not 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 is not 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.

$R — Real Size Directive

If $R2 is used the compiler produces a code file with 2-word real size. If $R4 is used the compiler produces a code file with 4-word real size. The Texas Instruments Professional Computer will default to 4-word real size. If you try to use 2-word real size you will receive error message 17 (Real size mismatch) during execution of the program.

$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 is not 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, and then in *SYSTEM.LIBRARY.

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

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

$U — User Program

This option is used to specify whether the compilation is a user 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, a user 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.

Conditional Compilation

You can conditionally compile portions of the source text. At the beginning of a program's text you can set a compile-time flag that 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 can 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 minus (−), that flag is set false. The flag can be followed by a plus (+) which sets it true. If no sign is present, a flag is true. The flag's name can also be followed by a caret (^) as shown below.

The state of a flag can be changed by a $D option that 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 can follow the flag's name with a minus (−), which causes the delimited code to be compiled if the flag is false. In the absence of a dash, the code is compiled if the flag is true. The flag's name can also be followed by a plus (+) or caret (^); these are ignored. In a $E option, the flag's name can be followed by a plus (+), minus (−), or caret (^); 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 a caret (^) yields the previous value of the flag. Each flag's stack can 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 a caret (^), the value returned is always false.

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

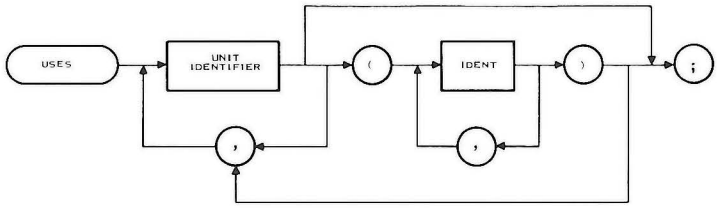
```
{$D DEBUG-} {declares DEBUG and sets it FALSE}
PROGRAM SIMPLE;
...
BEGIN
  {$D DEBUG+} {changes DEBUG to TRUE}
  ...
  {$B DEBUG} {if DEBUG is TRUE, this section is compiled}
  WRITELN('There is a bug.');
  {$E DEBUG} {this ends the section}
  ...
  ...
  {$D DEBUG^} {restores previous value of DEBUG}
              {... in this case, FALSE}
  {$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 does not 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:



2284068

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 is not 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 is not acceptable unless TYPE1 is added (even through 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. The first of these three compiled listings shows unit A, which is selectively used by units B and C.

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

  1  2  1:d   1    unit A;
  2  2  1:d   1    interface
  3  2  1:d   1    const
  4  2  1:d   1      maxnum=1000;
  5  2  1:d   1      maxchar=7;
  6  2  1:d   1
  7  2  1:d   1    type
  8  2  1:d   1      byte=0..255;
  9  2  1:d   1      codeblock=packed array
                    [0..maxnum] of byte;
 10  2  1:d   1      alpha=packed array
                    [0..maxchar] of char;
 11  2  1:d   1      ptr_info_rec=  info_rec;
 12  2  1:d   1      info_rec=record
 13  2  1:d   1               code:codeblock;
 14  2  1:d   1               llink,rlink:ptr_info_rec;
 15  2  1:d   1               end;
 16  2  1:d   1      next=char;
 17  2  1:d   1
 18  2  1:d   1    var
 19  2  1:d   1      first,last:byte;
 20  2  1:d   3
 21  2  1:d   3    function update(var info: ptr_info_rec)
                    :next;
 22  2  1:d   1
 23  2  1:d   1    implementation
 24  2  1:d   1
 25  2  1:d   1      function update;
 26  2  2:0   0      begin
 27  2  2:1   0        with info^do
 28  2  2:2   3        begin
 29  2  2:3   3          llink=rlink;
 30  2  2:3  12          if rlink=llink then
 31  2  2:4  22            update:='y'
 32  2  2:3  22          else
 33  2  2:4  27            update:='n';
 34  2  2:2  30        end;
 35  2  1:0   0      end;
 36  2  1:0   0
 37  2   :0   0    end. {unit A}

End of Compilation.
```

```
 1  2   1:d  1      unit B;
 2  2   1:d  1      interface
 3  2   1:d  1      {$U a.code}
 4  2   1:d  1      uses a( {const} maxchar,
                      {include for type ALPHA}
 5  2   1:d  1            {types} alpha,
                      {include for variable WHICH}
 6  2   1:d  1                  byte,
                      {include for FIRST and LAST}
 7  2   1:d  1            {vars } first,
                      {include for proc CHANGE}
 8  2   1:d  1                  last,
                      {include for proc CHANGE}

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

```
56  2   1:0  0              ptr_info_rec,
                {include for func UPDATE}
57  2   1:0  0                codeblock,
                {include for INFO_REC}
58  2   1:0  0                  next,
                {include for func UPDATE}
              Using A
59  2   1:0  0
61  2   1:u  1     maxnum=1000;
62  2   1:u  1     maxchar=7;
65  2   1:u  1     byte=0..255;
66  2   1:u  1     codeblock=packed array
                      [0..maxnum] of byte;
67  2   1:u  1     alpha=packed array
                      [0..maxchar] of char;
68  2   1:u  1     ptr_info_rec=^ info_rec;
69  2   1:u  1     info_rec=record
70  2   1:u  1          code:codeblock;
71  2   1:u  1          llink,rlink:ptr_info_rec;
72  2   1:u  1            end;
73  2   1:u  1     next=char;
78  2   1:u  1   function update
                    (var info:ptr_info_rec):next;
79  2   1:u  3
80  2   1:d  3       {func} update),
              Using B
81  2   1:d        b;
82  2   1:d
83  2   1:d      var
84  2   1:d        info:ptr_info_rec;
85  2   1:d        mine:alpha;
86  2   1:d  0
87  2   1:0  0   begin
88  2   1:1  0     new(info);
89  2   1:1  7     new(info^.rlink);
90  2   1:1 16     info^.llink:=nil;
91  2   1:1 22     mine:='newsystm';
92  2   1:1 30     if update(info)='y' then
93  2   1:2 39       writeln('info updated')
94  2   1:1 59     else
95  2   1:2 61       change(mine);
96  2    :0  0   end.
```

End of Compilation.

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 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 are not 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 can 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 can occupy a segment of its own; this is accomplished by declaring it a segment routine. Segment routines can be swapped independently of the main program; declaring a routine a segment is useful in managing main memory.

Routines that are not part of a program's main working set are prime candidates for occupying their own segment. Such routines include initialization and wrap-up procedures and routines that are used only once or only rarely while a program is executing. Reading a procedure in from a 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 which saves time and is less error prone. Libraries of correct routines can 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 Pascal host program. This may be necessary for performance reasons (assembly language is faster than p-code, the output of the compiler) 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 *UCSD p-System Assembler*, TI Part Number 2232402-0001.

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 display unit 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 do not want to compile a unit separately.

A unit consists of two main parts: the interface section, which can declare constants, types, variables, procedures, processes, and functions that are public (available to any client module); and 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 can reside when the client's code is executed:

- Within your code file
- In the SYSTEM.LIBRARY on the system disk
- In a user library
- In the operating system (SYSTEM.PASCAL)

The operating system units (described in the next chapter) are standard code. *Do not* 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 the Utilities chapter. Once the unit's code and your 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 are not 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 do not 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 clients 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 than 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 can reside in either of two locations:

- *SYSTEM.LIBRARY
- A code file specified in the text you are compiling

With UCSD Pascal, you indicate a specific code file using the $U (use library) compiler option which was described earlier. If you *do not* indicate a particular code file, the compiler searches *SYSTEM.LIBRARY for any units you want to use. If you *do* indicate a code file, the compiler looks there for the units. In the second case, if the unit is not found, the Pascal compiler searches *SYSTEM.LIBRARY, as well.

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 *do not* 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. In this second case, if the unit is not found, the Pascal Compiler searches *SYSTEM.LIBRARY, as well. However, the BASIC and FORTRAN compilers *only* search that particular code file. (See the individual reference manuals for more information about this.)

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: allowing separate compilation; and 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 the disk.

You can write a program with run-time memory management and separate compilations already planned, or you can write the program as a whole and then break it into segments and units. The latter approach is feasible when you are 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. 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: turning routines into segment routines; or 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.

3

User Interface

Introduction 3-3
Run-Time Application Facilities 3-3
The Screen Control Unit 3-6
SCREENOPS Interface Section 3-6
Routines Within SCREENOPS 3-8
Error Handler Unit 3-14
Format of Error Messages 3-14
User Control of Error Messages 3-15
The Command I/O Unit 3-17
Simple Color Interface 3-20
Turtlegraphics 3-22
The Turtle 3-23
The Display 3-27
Labels 3-28
Scaling 3-29
Figures and the Port 3-31
Pixels 3-34
Fotofiles 3-35
Routine Parameters 3-36
Sample Program 3-37

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 so 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.

Finally, a couple of facilities for using the color and graphics capabilities of your Texas Instruments Professional Computer are covered. The first is a simple interface that your programs can use to change the screen colors, and so forth. The second is a much more comprehensive package called Turtlegraphics.

RUN-TIME APPLICATION FACILITIES

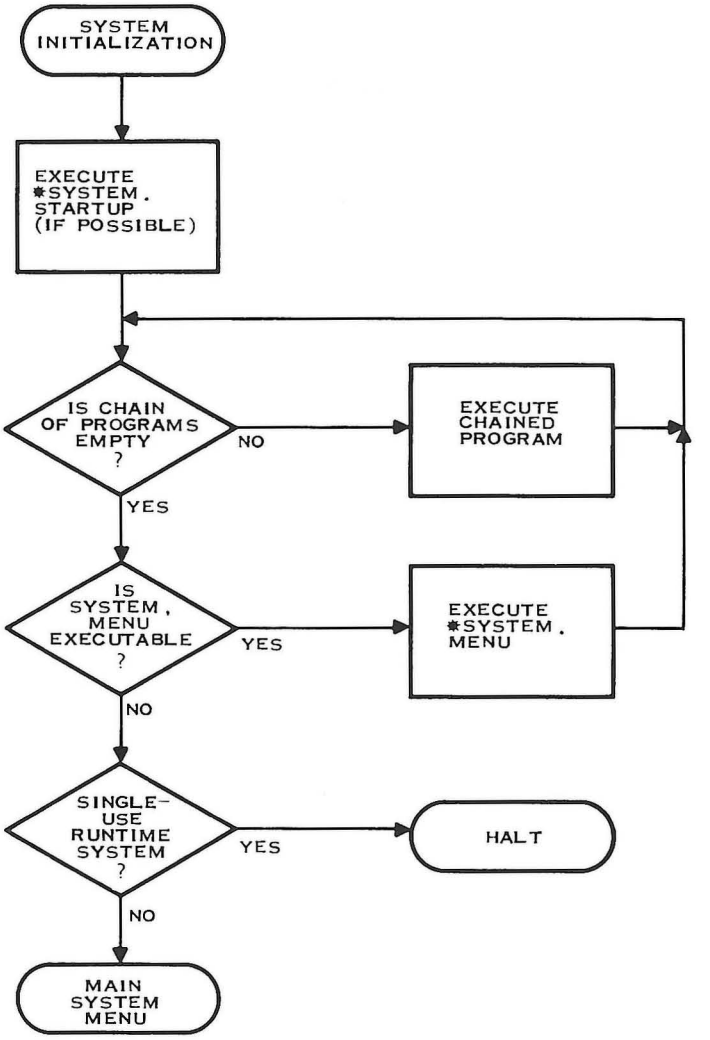
As an applications developer, you can create programs that are automatically executed by the p-System. This exempts the end-user from having to X(ecute these programs. The underlying p-System can even be completely hidden from such a user. You can present menus and prompts that apply specifically to 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 is normally 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 the user selects a component of your package, use the CHAIN procedure (within the operating system's Command I/O unit, described later in this chapter). CHAIN allows another program to execute (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 scenario, the p-System's command menu never appears.

The following diagram illustrates what happens when the p-System is initialized:


SYSTEM INITIALIZATION
EXECUTE *SYSTEM. STARTUP (IF POSSIBLE)
IS CHAIN OF PROGRAMS EMPTY ?
NO
EXECUTE CHAINED PROGRAM
YES
IS SYSTEM. MENU EXECUTABLE ?
YES
EXECUTE *SYSTEM. MENU
NO
SINGLE-USE RUNTIME SYSTEM ?
YES
HALT
NO
MAIN SYSTEM MENU

2284069

THE SCREEN CONTROL UNIT

The screen control unit is a unit within the operating system that 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 display units.

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 do not 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 a 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 (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 is not 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.

Function SC_Find_Y: integer;

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

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

SC_ChSet is a SET OF CHAR. This procedure repeatedly reads from the keyboard into CH until CH is equal to a member of Return_on_Match. The characters that you pass in this set should all be capitals (if they are alphabetic). If a 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 was not read. After reading a space successfully, this function executes 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 pressing a question mark (?). 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 function value of the character received.) The keyboard is repeatedly read until the character read matches one within the Return_on_Match set. (All alphabetic characters in this set should be capitals.)

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

While a string is being read, this function can 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, SC_Insert_Key, SC_Delete_Key). The character passed is mapped into one of these elements. SC_Not_Legal is where all characters are mapped that do not fit into one of the other ten categories. Prefix characters are recognized by this function. If you pass a prefix character, a non-echoed 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, yet another read can 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 keyboard 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 is not in the appropriate drive, you are asked to replace the disk and press the space bar to continue. If a program attempts to divide by zero or access outside the bounds of a Pascal array, a message indicates this and the user is asked to press the space bar, 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 is not 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 was not found on the volume in device U. Place the volume VOLNAME in the correct drive and press the space bar. Execution should continue normally.

The following example shows the error message that occurs when a user presses the p-System **BRK** (Break) key. BRK is a shifted version of the BRK/PAUS key.

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

After the space bar is pressed, the p-System is reinitialized.

System error messages, such as these, always appear at a fixed position on the display unit. 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 the space bar, 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. This is because the p-System is not 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 is not present in the appropriate drive for blocked devices. If the code file is on a subsidiary volume, set the message for the principal volume.

The ERRORHANDLING unit provides these facilities. The file ERRORHAND.CODE contains this unit. To use ERRORHANDLING, a Pascal program should have a declaration similar to the following example.

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

Also, ERRORHANDLING must be available at 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 can change the standard message for code segments needed on disks that are not present. Call 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 the user program has designated a special error message, that message is displayed. The p-System then waits for the user to press the space bar, 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

A user message is destroyed if a MARK is called before a SET_USER_MESSAGE.

NOTE

The physical device numbers are #4, #5, and #9 through the maximum number for physical disks 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 the user to press the space bar. 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 the ESC key.

CAUTION

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

Error message values that 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 display unit 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: manually pressing X to call X(ecute; and 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 the user. 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, the user's program should follow it with a call to EXCEPTION in order to turnoff all redirection. If the user does not 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 would leave the system in an indeterminate state).

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

SIMPLE COLOR INTERFACE

The p-System on the Texas Instruments Professional Computer provides a simple interface to assist you in utilizing the special features of the color and grey scale display unit. The following table defines the character attribute set:

Character Attributes Set

Attribute	Description
COLOR/INTENSITY	Defines the color of subsequent output when the color display unit is installed or the intensity of output when the gray scale display unit is used (default white)
ENABLE	Defines whether or not subsequent output will be displayed on the display unit (default display enabled)
REVERSE VIDEO	Defines whether background or foreground is to be highlighted (default not underlined)
BLINK	Defines whether characters displayed will be underlined (default not underlined)
ALTERNATE CHARACTER SET	Defines whether character fonts will be defined by the primary or alternate character set (default primary)

NOTE

You should realize that more than one attribute may be specified but that certain combinations will not make sense. For example, if character enable is false, then output does not appear nor do any of the other attributes except for reverse video.

The next table lists special characters that you can send to the display unit using the UNITWRITE intrinsic from a Pascal program. (You should see *UCSD Pascal*, TI Part Number 2232401-0001, for information about UNITWRITE.) Each of these codes must be preceded (or *prefixed*) by the ASCII ESC character, decimal 27.

Attribute Character Sequences

Character	Attribute Toggled	Description
81	GLOBAL Enable	Changes the attribute of each displayed character to the most recently defined attribute.
82	BLACK Enable	Enables BLACK (no output)
83	BLUE Enable	Enables BLUE output
84	RED Enable	Enables RED output
85	GREEN Enable	Enables GREEN output
86	CYAN Enable	Enables BLUE-GREEN output
87	YELLOW Enable	Enables RED-GREEN output
88	MAGENTA Enable	Enables BLUE-RED output
89	WHITE Enable	Enables WHITE output
90	CHARACTER Enable	When character enable is active (true) characters are displayed on the display unit when output. When false, characters will not be displayed on the display unit
91	REVERSE VIDEO TOGGLE	Toggles REVERSE VIDEO background/foreground
92	UNDERLINE TOGGLE	Toggles UNDERLINING of characters displayed
93	BLINK TOGGLE	Toggles BLINKING of characters displayed
94	ALTERNATE CHAR SET	Enables/disables the ALT character set for output
95	GLOBAL RESET	RESETs all attributes to the system defaults

Sequences that enable functions OR an attribute with the previous attribute set. Sequences that toggle functions XOR an attribute with the previous value of the attribute and then OR the new attribute value into the attribute set. Simply put, toggles turn attributes on the first time they are output and off the next time they are output. Sequences that enable attributes always turn those attributes on. Defining a new attribute does not affect the remainder of the attribute set (that is, changing color does not affect whether or not the character is displayed with an underline).

Although these sequences provide a simple interface to the color capabilities of the high resolution color graphics monitor, the Turtlegraphics package, described later, is available to provide more comprehensive access to the capabilities of the system console.

There are some special keys described in Appendix A which are related to display intensity, color, cursor appearance, and so forth.

TURTLEGRAPHICS

Turtlegraphics is a package of routines for creating and manipulating images on your Texas Instruments Professional Computer's display unit. These routines can be used to control the background of the display unit, draw figures, alter old figures, and display figures using viewports and scaling. Turtlegraphics 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, such as the display unit. Other figures can be saved in memory. Each figure has a turtle of its own. You can set the size of a figure (it does not need to be the same size as the actual display).

The actual display is addressed in terms of a display scale, which can 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 can also define a viewport, or window on the display. This limits all graphic activity to within that port.

In order to use Turtlegraphics, your Pascal programs must include a uses statement like this:

```
USES Turtlegraphics;
```

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

The Turtle

The turtle is an imaginary creature in the display screen that draws lines as your program moves it around the display. The turtle can move in a straight line (Move), move to a particular point on the display (Moveto), turn relative to the current direction (Turn), and turn to a particular direction (Turnto).

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

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

NOTE

The turtle may be moved anywhere; it is not limited by the size of the figure or the size of the display. But, only movements within the figure are 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 is visible unless the current turtle is in a figure that is not currently on the display.

Procedure Moveto (x,y: real);

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

Procedure Turn (rotation: real);

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

Procedure Turnto (heading: real);

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

Procedure Pen_color (shade: integer);

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

Here are the colors that correspond to the shade number:

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

The term wild card refers to the standard background color of your display. This depends on your display hardware, and might be called a *hard* background (you may or may not be able to change it from a program; this depends on your hardware configuration). In Turtlegraphics, each individual figure may have its own *soft* background color, which we refer to simply as the *background color* (as in the discussion below).

You may also use black and white graphics, in which case the colors might be simply:

0 = Black
1 = White

Procedure Pen_mode (mode: integer);

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

These are the possible modes:

0 = Nop — does not alter the figure.

1 = Substitute — writes the current pen color.

2 = Overwrite — writes the current pen color.

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

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

Values greater than four are treated as Nop.

These descriptions apply to movements of the turtle. They have a more complex meaning when a figure is copied onto a figure that is already displayed.

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

Function Turtle_y : real;

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

Function Turtle_angle : real;

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

Procedure Activate_Turtle
(screen: integer);

Specifies to which figure subsequent Turtlegraphics commands are directed. Each invocation of this procedure puts the previously active turtle to sleep and awakens the turtle in the designated figure. When Turtlegraphics is initialized, the turtle in the actual display is awake. The initial position of the turtle is (0,0) or the bottom left corner of the display unit, 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 the drawing mode is used, as indicated above.

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

NOTE

If you use Turtlegraphics (or customized routines of your own) to alter the settings of your display, it is a good idea to reset everything before your program terminates. Usually it is not possible for the display to return to its original state, and the p-System software has no knowledge of what that original state was.

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

Fills the specified figure (*screen*) with the specified color (*shade*). If screen = 0, which indicates the actual display unit, then only the current viewport is shaded. For user-created figures, the entire figure is shaded.

Procedure Background
(screen: integer; shade: integer);

Specifies the background color for a figure. The initial background color of all figures is the 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.

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 the user-written procedure Query_Environment).

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 display unit. 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 insure that the following ratio is the same as the aspect ratio of the physical display unit upon which the image is displayed.

```
(max_x − min_x) / (max_y − min_y)
```

When the Turtlegraphics unit is initialized, min_x and min_y are set to zero. 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, use a call similar to the following example.

```
Display_scale(0, 0, 100*ASPECT_RATIO, 100);
```

This utilizes Function Aspect_ratio described above, and makes the y axis 100 units long.

Turtlegraphics always treats the turtle as being in a fixed pixel location. Changing the scaling of the system with a call to this routine in the middle of a program does not alter the pixel position of any of the turtles in the figures. However, the values returned from X_pos and Y_pos may change.

Figures and the Port

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 display unit (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 display unit in which all subsequent graphics activity takes place. You can create a figure, specify the port, then display that figure (or a portion of it) within the port. Specifying a viewport does not restrict turtle activity, it merely restricts what is displayed on the display unit.

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 can be created using Create_Figure (or it can 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 unit (the destination) using the drawing mode specified. The figure is placed on the display such that its lower left corner is at (corner_x, corner_y). The x and y positions are specified in the units of the current display scale. If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

The following items define the drawing mode numbers.

0 Nop. Does not alter the destination.

1 Substitute. Each pixel in the source replaces the corresponding pixel in the destination.

2 Overwrite. Each pixel in the source that is not of the source's background color replaces the corresponding pixel in the destination.

3 Underwrite. Each pixel in the source that is not of the source's background color is copied to the corresponding pixel in the destination only if the corresponding pixel is of the destination's background color.

4 Complement. For each pixel in the source that is not of the source's background color, the corresponding pixel in the destination is complemented.

Values greater than four 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 unit 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 is covered when the figure is placed on the display with its lower left-hand corner at (corner_x, corner_y). If the display scale has been modified since the figure was created, the results of this procedure are unpredictable.

NOTE

When a figure is moved to the display by 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, you must 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 are not 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: integer);
```

Sets the pixel at the x,y location of the specified figure to the specified color. The x,y location is specified in the units of the current display scale.

Fotofiles

You can create disk files that contain Turtlegraphics figures. New figures can be written to a file, and old figures restored for viewing or modification.

When figures are written to a file, they are written sequentially, and assigned an *index* that is their location in the file. They may be retrieved randomly by using this index value.

The p-System name for files of figures always contains the suffix *.FOTO.* It is not necessary to use this suffix when calling Read_figure_file or Write_figure_file (if absent, it is supplied automatically).

```
Function Read_figure_file
    (title: string): integer;
```

Specifies the title of a file from which all subsequent figures are loaded. If a figure file is already open for reading when this function is called, it is closed before the new file is opened. Only one figure file may be open for reading at a single time. This function returns an integer value which is the ioresult of opening the file.

```
Function Write_figure_file
    (title: string): integer;
```

Creates an output file into which user-created figures may be stored. If another figure file is open for writing when this function is called, it is closed, with lock, before the new file is created. Only one figure file may be open for writing at a single time. This function returns an integer result which is the ioresult of the file creation.

Function Load_figure
    (index: integer): integer;

Loads the indexed figure from the current input figure file and assigns it a new, unique, figure number. An automatic Create_figure is performed. If the operation fails for any reason, a Figure_number of zero (0) is returned.

Function Store_figure
    (figure: integer): integer;

Sequentially writes the designated figure to the output figure file. The function returns an integer that is the figure's positional index in the current output figure file. Positional indexes start at one (1). If the index returned equals zero (0), Turtlegraphics did not successfully store the figure.

Routine Parameters

The next example 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 );
procedureViewport( min_x, min_y, max_x,
                max_y : real );
```

```
procedure Fillscreen( screen:
                integer ; shade:
                integer );
procedure Background (screen: integer;
                      shade : integer );
function Read_pixel( screen: integer ;
                x, y : real ) :integer ;
procedure Set_pixel( screen:integer ;
                x,y:real ; shade:color );
procedure Get_Figure( source_screen:
                integer ,
                corner_x, corner_y: real ;
                mode : integer );
procedure Put_Figure( destination_screen:
                integer ,
                corner_x, corner_y: real ;
                mode : integer );
function Read_figure_file( title : string ):
                integer ;
function Write_figure_file( title : string ):
                integer ;
function Load_figure( index : integer ):
                integer ;
function Store_figure( figure: integer ):
                integer ;
procedure Activate_Turtle( screen:
                integer );
function Turtle_x : real ;
function Turtle_y : real ;
function Turtle_angle : real ;
procedure Move( distance : real );
procedure Moveto( x,y : real );
procedure Turn( rotation : real );
procedure Turnto( heading : real );
procedure Pen_mode( mode : integer );
procedure Pen_color( shade : integer );
procedure WChar( c: char ; copymode, shade: integer);
procedure WString( s: string; copymode, shade: integer);
```

Sample Program

Here is a sample program that illustrates a number of Turtlegraphics routines:

```
program Spiraldemo;

  uses Turtlegraphics;

  const nop = 0;
    substitute = 1;
```

```
var I, J, Mode: integer ;
    C: char ;
    Color: integer ;
    Seed: integer ;
    LX, LY, UX, UY: real ;

function Random (Range: integer ): integer ;
  begin
    Seed:= Seed * 233 + 113;
    Random:= Seed mod Range;
    Seed:= Seed mod 256;
  end ;

procedure ClearBottom;
    {clears bottom line of screen
     for prompts}
    begin
      Penmode (nop);
      Moveto (0, 0);
      WString ('                          ', substitute, 1);
  end ;
begin
  ClearBottom;      {various initializations}
  WString ('ENTER RANDOM NUMBER: ', substitute, 1);
  read (keyboard , Seed);
  ClearBottom;
  Display_Scale (0, 0, 200*Aspect_Ratio, 200);
        {Aspect_Ratio used so
        pattern will be round}
  Color:= 0;
  WString ('ENTER VIEWPORT LL CORNER: ', substitute, 1);
  read (keyboard , LX,LY);
  ClearBottom;
  WString ('ENTER VIEWPORT UR CORNER: ', substitute, 1);
  read (keyboard , UX,UY);
  ClearBottom;
  WString ('PENMODE= ', substitute, 1);
  read (keyboard , MODE);
  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 corner}
GetFigure (I, 0, 0, 1);
{display finished spiral}
readln ;
    {clear user input buffer}
end .
```

4

File Management Units

Introduction 4-3
Interface Sections 4-4
Directory Information 4-10
Notation and Terminology 4-11
File Name Arguments 4-12
File Type Selection 4-13
File Dates 4-15
Error Results 4-15
The DIR_INFO Routines 4-16
D_PINFO 4-26
Function Result 4-28
Wild Card File Name Change 4-35
Wild Cards (WILD) 4-49
Special Wild Card Characters 4-49
Question Mark Wild Card 4-50
Equals Sign Wild Card 4-50
Subrange Wild Card 4-51
D_Wild_Match Parameters 4-53
D_Wild_Match Pattern Matching Info 4-53
System Information (SYS.INFO) 4-57
File Information (FILE.INFO) 4-61
Type F_File_Type = file; 4-61
Time Date Unit 4-63

INTRODUCTION

Your Pascal programs on the Texas Instruments Professional Computer 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 an 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 are not 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 wildcard 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 wildcards) and returns true if they match. Includes
information about pattern matching that occurred if re quested (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 }
                            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 (wildcard 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 wildcards }

Function D__Rem__Files (D__Name : String; D__Select : D__Choice) : D__Result;
{ Removes file of specified name (wildcards allowed) }

Procedure D__Lock;
Procedure D__Release;
{ Provide means to limit use of DirInfo routines to one task at a time in multi-tasking
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 workfile code }

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

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

  Procedure SI_Text_Tid (Var SI_Title : String);
    { Returns title of current workfile 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 File_Info;

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
  block-structured unit. If there is no file opened for the Fid or if the device is not block
  structured, the function result is set to false.}

Procedure F_Date (Var Fid: F_File_Type;
                  Var File_Date: F_Dat_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.

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

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 *UCSD p-System Operating System Reference Manual*, TI Part Number 2232395-0001, for a complete description of UCSD p-System files and file names if you are not familiar with them.

Volume names and file names can 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 UCSD 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 is not a valid file name in the p-System.

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

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 cannot 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 can 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__DateRec = Packed Record
                Month : 0..12;
                Day : 0..31;
                Year : 0..100;
             End;{ D__DateRec }
```

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

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 is not 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 cannot 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 never causes a function to terminate abnormally. Errors that the routine cannot identify explicitly 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).

CAUTION

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 UCSD 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 does not determine whether or not D_Name actually exists.

D_Scan_Title accepts the following parameters.

- D__NAME—A string containing a UCSD 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 do not contain blank characters.

- D__TITLE—A string that returns the file name contained in D__NAME. If D__NAME does not contain a file name, D__TITLE is assigned the empty string. File titles assigned to D__TITLE contain only uppercase characters and do not contain blank characters.

- D__TYPE—A scalar which returns a value indicating the file type of the file name contained in D__NAME.

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 is not .CODE.

D__SEGS—An integer that is assigned a value indicating the presence of a file length specifier in D__NAME. The value returned in D__SEGS is assigned as follows:

LENGTH SPECIFIER	D__SEGS VALUE
[< number>]	< number>
[*]	−1
< not present>	0

D__Scan__Title returns a function result of type D__Result. The only scalar values returned by D__Scan__Title are D__Okay and D__Name__Error; they have the following meanings:

- D__Okay—No Error. All information returned by D__Scan__Title is valid.
- D__Name__Error—Illegal file name syntax in D__NAME. The information returned by D__Scan__Title is invalid.

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, numbers 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 can 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 is not requested.

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

- D__IsBlkd is set to true if the file object is (or resides on) a 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, 5¼-inch 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 is not requested or if the file ID in D__NAME is empty.

- D__Date contains the file date for the current object.

- D__NumFiles is valid only for objects of type D__Vol; it contains the number of files in the volume's directory.

NOTE

An .SVol file (which contains a subsidiary volume) appears as any other file on the principal volume. This means that D__NumFiles does not correspond to an .SVol file. However, 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, CompPos is 5
   ('=' matches '.CODE')

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

NOTE

If the volume ID in D_NAME consists of a 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, CompPos is 4
   ('#5' matches 'MYDISK')

NOTE

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

Function Result

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

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

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

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

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

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

Example Program

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

```
Program Listtest;
Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  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;
    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 { D_Test }
  Repeat
    Mark (Heap_Ptr);
  Select := [ ];
    Writeln ('Directory Lister - ');
    Write ('Volume and/or file name to match: ');
    Readln (Match);
    Write ('Return pattern matching information? [y/n] ');
    Read (Ch); Writeln ;
  Want_Patterns := Ch In ['y', 'Y'];
  If Want_Patterns Then
  Result := D_ScanTitle(Match, Volume, Title, Typ, Segs);
    Writeln ('Types [ y/n ] : ');
    GiveChoice('Directories', [D_Vol]);
    GiveChoice('Text Files ', [D_Text]);
    GiveChoice('Code Files ', [D_Code]);
    GiveChoice('Data Files ', [D_Data]);
    GiveChoice('Temp Files ', [D_Temp]);
    GiveChoice('Free Space ', [D_Free]);
    GiveChoice('SVol Files ', [D_SVol]);
    Result := D_DirList(Match, Select, Ptr, Want_Patterns);
    Writeln ;
    If Ptr <> Nil Then
    Print_Info(Ptr)
    Else
      Case Result Of
        D_Name_Error : Writeln ('  Error in file name');
        D_Off_Line : Writeln ('  Volume off line');
        D_Not_Found : Writeln ('  File not found');
        D_Other : Writeln ('  Miscellaneous error');
    End; {cases}
    Writeln ;
    Repeat
            Write ('Continue ? ');
            Read (Ch); Writeln ;
        Until Ch In ['n','N','y','Y'];
    Writeln ;
    Release (Heap_Ptr);
    Until Ch In ['n', 'N'];
    End. { listtest }
```

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

D__Change__Name searches for the volume or file designated by the file name contained in D__OLD__NAME and changes its name to the file name contained in D__NEW__NAME.

D__Change__Name only changes one file name at a time, and thus does not accept file names containing wild cards; however, it can be combined with other Dir__Info and wild card routines to create user-defined file name changing routines 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 can 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 was not on-line. No change made.

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

For example, assume that 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 catenation of the pattern matched by the = and the literal string A. The part of each file title matched by the period and the second = wild card is unchanged. D_Change_Name is called with the modified file title for each file matched by D_Dir_List.

Example Program

The following program demonstrates how you can 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 WildChng;

Uses
  (*$UWILD.CODE*)
  wild,
  (*$UDIR.INFO.CODE*)
  DirInfo;
Var
  Heap_Ptr : ^Integer;
  Typ : D_NameTyp;
  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 ThenWrite('^':(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 <>) 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 is not 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 do not contain file dates. Temporary status for files is specified by a special value in the file date field. Thus, D__Free and D__Temp are ignored if they are included in D__SELECT.

D__Change__Date returns a value of type D__Result. D__Change__Date can return all scalar values defined in D__Result except D__Exists; the values 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 was not on-line. No change made. This error occurs only if the volume ID in D__NAME specifies a single volume which is off-line. If the volume name in D__NAME contains wild cards and does not match any on-line volumes, D__Change__Date returns D__Not__Found.

- D__Other—Unknown error. No change made. D__Change__Date encountered an unidentified error which prevented successful completion of the operation.

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 -');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__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 can 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 can 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 cannot be removed from the directory; thus, D__Free is ignored if it is included in D__SELECT.

D__Rem__Files returns a value of type D__Result. D__Rem__Files can return all scalar values defined in D__Result except D__Exists; the values have the following meanings.

- D__Okay—No error. D__NAME was found. If D__Vol is included in D__SELECT, and a volume matches the file name argument in D__NAME, the volume is taken off-line. If D__Text, D__Code, D__Data, D__SVol, or D__Temp are included in D__SELECT, disk files of those types which match D__NAME are deleted from their directories.

- D__Not__Found—No such file/volume found. No match found for D__NAME. No change made.

- D__Name__Error—Illegal file name syntax in D__NAME. No change made.

- D__Off__Line—Volume off-line. The volume specified by D__NAME was not on-line. No change made. This error occurs only if the volume ID in D__NAME specifies a single volume which is off-line. If the volume ID in D__NAME contains wild cards, but does not match any on-line volume, D__Rem__Files returns D__Not__Found.

- D__Other—Unknown error. No change made. D__rem__Files encountered an unidentified error which prevented successful completion of the operation.

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

Procedure D_Lock

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

NOTE

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

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

Procedure D__Release

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

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 p-System manual that describes the filer operation. Because of the extra functions provided by this UNIT, there is not a direct correspondence between the filer and this UNIT. Where there are differences in the use of characters, these are described.

Special Wild Card Characters

The following characters are defined as special characters:

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

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

Examples of percent sign in wild cards:

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

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

Equals Sign Wild Card

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

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

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 preceded by a tilde is entirely excluded from the character set. The list of character items is evaluated left to right. Characters specified by nonnegated 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 nonnegated items, but does not match any of the negated choices. For example, the subrange {*a*—*z*~*r*} represents the set of characters from *a* to *z*, excluding *r*.

NOTE

Blank characters within subrange wild cards are ignored. Wild card characters can be specified in character sets with the percent sign notation described 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 can 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 do not match, or the list was not 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 : ');
    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 }
```

SYSTEM INFORMATION (SYS.INFO)

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 a user requires a different set, then another unit similar to this one can be easily constructed for the particular situation.

In order to distinguish the variables defined by this unit, they have been prefixed with SI. Here are the SYS.INFO routines:

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 will not change the date on the system disk. It will only change the date internally in the operating system. To change the date on the disk, use function D_Change_Date within the DIR.INFO unit.

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

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

```
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 (FILE.INFO)

This unit provides an easy way to access information in the file information block (fib). It uses the system globals from KERNEL.CODE. Although it is possible for you to access this global data, it is easier to use this unit. In order to distinguish the 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 passed on as parameters to these procedures.

```
Function F_Open
    (var fid: F_File_Type):boolean;
```

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

```
Function F_Length
    (Var Fid : F_File_Type) : Integer;
```

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

TIME DATE UNIT

This unit allows a program to read or set the hardware internal date and time. (This is an extension to the p-System for the TI PC.) This unit is located in the system library and is referenced by the UCSD Pascal USES declaration. The unit requires six parameters which are as follows:

Parameters	Range
HOURS	0-23
MINUTES	0-59
SECONDS	0-59
HUNDREDTHS OF SECONDS	0-99
DATE	0-9999
FLAG	0-3

The time is kept in 24-hour format and date is simply a count of days since the clock was started. This date is not the same as the p-System date. When the system is booted, the time is set to 00:00:00:00 and the date is set to 0.

There is no range checking of values passed to the unit. Therefore, each parameter value must meet the range criteria listed above or results are unpredictable. The date is kept and updated by the Settime utility as the Julian date (1 . . 366). The system increments the date field by one every 24 hours.

The procedure TIMEDATE(HOURS, MINUTES, SECONDS, HUNDSECS, DATE, FLAG), is used to read and set the time and date. This is accomplished by setting the FLAG parameter to one of the following values:

GET TIME = 0
SET TIME = 1
GET DATE = 2
SET DATE = 3

The following example demonstrates calling the Time/Date unit.

```
PROGRAM EXAMPLE;
USES SYSTIMDT;
{  example of how to use system time/date unit  }

Var Hr, Min, Sec,
      Hsec, Date, Flg : Integer;

Begin  {  example  }

Hr              :=  9;
Min             :=  30;
Sec             :=  0;
Hsec            :=  0;
Date            :=  0;
Flg             :=  1;
              {  set flag to set time  }

{  call TimeDate to set time  }
TIMEDATE(Hr, Min, Sec, Hsec, Date, Flg);
```

```
{ call TimeDate to get time }
Flg          := 0;                  { set flag to get time }
TIMEDATE(Hr, Min, Sec, Hsec, Date, Flg);
Writeln('Time:  ',Hr,':', Min, ':', Sec, '.',Hsec);

{ call TimeDate to set date }
Date     := 30                  { set to day of month }
Flg      := 3;                    { set flag to set date }
TIMEDATE(Hr, Min, Sec, Hsec, Date, Flg);

{ call TimeDate to get date }
Date     := 0;                    { clear date parameter }
Flg      := 2;                      { set flag to get date }
TIMEDATE(Hr, Min, Sec, Hsec, Date, Flg);
Writeln('Date: ',Date);

End.  { example }
```

5

Debugging and Analysis

Introduction 5-3
Using the Debugger 5-3
Entering and Exiting 5-4
Using Breakpoints 5-5
Viewing and Altering Variables 5-6
Viewing Text Files 5-8
Displaying Useful Information 5-9
Disassembling p-Code 5-10
Example of Debugger Usage 5-11
Symbolic Debugging 5-12
Symbolic Debugging Example 5-14
Summary of the Commands 5-17

INTRODUCTION

The symbolic debugger is a tool for locating and correcting errors that might exist in your compiled programs on the Texas Instruments Professional Computer. You can call it from the command menu. It can also be invoked while a program is running (when a breakpoint is encountered). Using the symbolic debugger, you can 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 UCSD p-machine architecture and understand the p-code operators, stack usage, variable and parameter allocation, and so on. These topics are discussed in the *UCSD p-System Internal Architecture*, TI Part Number 2232400-0001.

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 the space bar 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 was not previously active, and no breakpoints are currently enabled. If you enter the debugger in a nonfresh state, only the left parenthesis (appears.

Exit the debugger by pressing Q to call Q(uit, R to call R(esume, or S to call S(tep. The Q(uit option disables the debugger. If the debugger is called again, it returns in a fresh state. The R(esume option will not 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 Breakpoints

To enter the debugger while a program is running, but not alter the program's code, use the debugger to set breakpoints. Press B to call the B(reakpoint option and then use either the S(et, R(emove, or L(ist command. To set a breakpoint, press S (et after pressing B(reakpoint. There are, at most, five breakpoints numbered zero through four. 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 the space bar. The next prompt is:

```
Segname?
```

Enter the name of the desired segment and press the space bar. The next prompt is:

```
Procname or #?
```

Enter the number of the desired procedure and press the space bar. The final prompt is:

```
Offset #?
```

Enter the desired offset within the procedure and press the space bar. The system sets a breakpoint; 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 breakpoint. If no compiled listing is available, use the text file viewing facility.

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

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

```
Remove break #?
```

To remove a breakpoint, enter its number; then press the space bar.

To list the current breakpoints, 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.

```
(1) S=INIT P#1 V0#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 HEX CODE and then in ASCII (a dash (−) indicates that the character is not a printable ASCII character).

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

The eight bytes that are currently displayed can be altered. If a / is pressed, then the line can be altered in hexadecimal mode. If a \ #92 is pressed, then the line can be altered in ASCII mode. When altering in hexadecimal mode, any characters that are to be left unchanged can be skipped by pressing the space bar. In the ASCII mode, any characters to be left unchanged can be skipped by pressing the **RETURN** key.

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, beginning with the most recently created one. 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 pressing the space bar. 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 is not available. This command enables you to 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 breakpoint is encountered), it displays various information if it is desired. This information includes p-machine registers, the current p-code operator, the information in the current markstack, or any specified memory location. In order to select which information is displayed, use the E(nable mode. After pressing 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 O#23 2C1A: 0B 05 53 43 41 4C 43 61 ----SCALCa
```

To initialize this address to a given value, use A(ddress mode at the outer level. Press A(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 can be exited before it reaches the ending offset by pressing the **BRK/PAUSE** key (BRK is the upper case version of this key); control returns to the debugger.

Example of Debugger Usage

Suppose the following program is to be debugged:

```
Pascal Compiler IV.0

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

End of Compilation.
```

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

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

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

```
Hit break #0 at S=NOTDEBUG P#1 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,* and *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 do an integer comparison (<>). This is where the error appears; so 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 ...
```

You list the registers and then look at the memory address to which register sp points. You 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 nonsense. This leads you to suspect that K was not initialized. Looking over the listing, you 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, breakpoints and portions of code to be disassembled can be indicated by procedure name and line number, rather than procedure number and p-code offset.

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.

Using symbolic debugging, breakpoints can 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 breakpoint.

Variables within a given routine can 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 are not allowed to be specified symbolically. The V(ar command prompts:

```
Varname or Offset #?
```

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

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

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

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

```
Procname or #?
```

Enter the first eight characters of the procedure name. The system displays the following prompt:

```
First #____ Last #____ Start Line#? 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
(1) S=EXAMPLE P=SETCIFD V=D E7B2 : 0000 9448 BEE7 190C-H-
(Q)
```

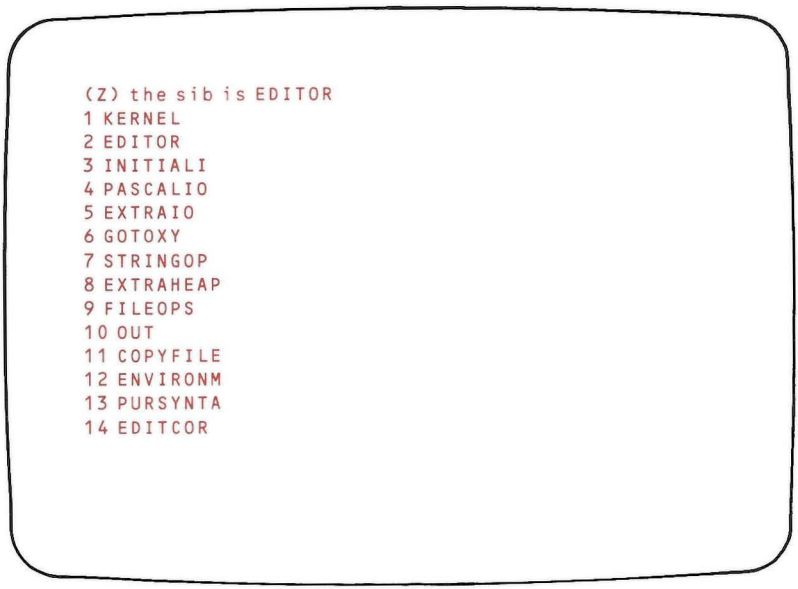
The first time the debugger is entered, the program example is not in memory and, hence, the symbolic segment is not in memory. However, a breakpoint can still be set symbolically providing you know on which line number to stop. For the second breakpoint, the symbolic segment is in memory; because of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

NOTE

> 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 is not 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 lists all of the principal segments with their environment list. For example, the following example is a partial list of the principal segments (EDITOR and EXAMPLE) and their environments. The lowercase name example is the symbolic segment for EXAMPLE. The existence of example indicates that symbolic debugging information is available for at least one procedure in EXAMPLE.



SUMMARY OF THE COMMANDS

A(ddress	Displays a given address.
B(reakpoint	Segment, procedure and offset must be specified.
S(et	Allows a breakpoint (0 through 4) to be set.
R(emove	Allows a breakpoint to be removed.
L(ist	Lists current breakpoints.
C(hain	Changes frame of reference for V(ariable command.
U(p	Chains up mark stack links.
D(own	Chains down mark stack links.
L(ist	Lists current mark stacks.
F(ile	Allows viewing of text files.
E(nable	Enables the following to be displayed.
D(isable	Disables the following from being displayed.
L(ist	Lists the following.
R(egister	The registers: mp, sp, erec, seg, ipc, tib, rdyq.
P(code	Current p-code mnemonic.
M(arkstack	Mark stack display.
A(ddress	A given address.
E(very	All of the above.
I(nteractive	Interacts with the performance monitor.

M(emory	
L(ock	Memlocks the debugger.
S(wap	Memswaps the debugger.
P(code	Disassembles 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 segment of given procedure.
E(xtended	Displays variables in another segment.
Z(seg list	Displays segment lists.

6

Utilities

Introduction 6-3
Decode 6-3
DECODE Programming Example 6-5
D(ictionary Display 6-8
Disassembled Listing 6-9
The Library Utility 6-12
Using Library 6-13
Library Example 6-14
Native Code Generator 6-17
Native Code Directives and Pascal 6-18
Running the Native Code Generator 6-18
Limits of Native Code Generation 6-20
Patch 6-21
EDIT Mode 6-22
TYPE Mode 6-23
DUMP Mode 6-25
Prompts 6-27
Print Spooling 6-28
REALCONV Utility 6-29
XREF — the Cross-Referencer 6-31
Introduction 6-31
Referencer's Output 6-32
Lexical Structure Table 6-32
The Call Structure Table 6-33
The Procedure Call Table 6-34
Variable Reference Table 6-34
Variable Call Table 6-34
Warnings File 6-35
Using Referencer 6-36
Limitations 6-38

INTRODUCTION

The UCSD p-System's utilities are various precompiled programs that can assist you in various 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 Library utility which is used to place separately compiled units into the client's code file or into library files.
- The Native Code Generator which converts portions of a p-code file into machine code.
- The Patch utility which enables you to view the internal content of any sort of file.
- The Print Spooler utility which allows you to print files as you are using the p-System normally.
- The Real Convert utility which can improve the performance of large programs which use several real constants.
- The XREF utility which is useful for analyzing Pascal programs.

DECODE

The decoder 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 *UCSD p-System Internal Architecture*, 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 will not search the disk for UNITs to decode. Assembly routines linked into a higher-level host will not be disassembled when the host is decoded.

When 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 DECODEs output may be written. This may be CONSOLE: (indicated by pressing the **RETURN** key), REMOUT:, PRINTER:, or a disk file. The system then displays the following menu:

```
Segment Guide: A(11), #(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 pressing the **RETURN** key 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	HSG	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:  1  Block offset:    26  Seg offset:  26
Data size:          0       Exit IC:  38
          Offset                                Hex code
          0(000):   LDCB      50                  8032
          2(002):   SRO        1                  A501
          4(004):   SCXGADDI             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       Exit 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 displays the code file segment dictionary. The following items describe the information that is displayed.

Inx (Index)	DECODEs name for each segment; individual segments may be disassembled by entering their number and pressing the RETURN key; for example, pressing 0 and then pressing the RETURN key 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 UCSD p-System version number of the segment.

M_TYPE is the machine type. Usually this is M_PSEUDO, indicating a p-code segment, but assembled segments 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 display unit.

On the same line, the block number of next portion of the dictionary is displayed after *Next Page:*. (In this example, the segment dictionary is entirely contained in block zero so the 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 zero.) The block offset and segment offset are always zero. The information occupies the first 11 words (0 through 10) of the segment. This housekeeping information (which is described in the *UCSD p-System 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 *UCSD p-System 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(11), #(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 pressing the RETURN key, 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
```

THE LIBRARY UTILITY

LIBRARY.CODE is a utility program that allows you to group separate compilations (units or programs) and separately assembled routines into a single file. A library is a concatenation of such compilations and routines. Libraries are a useful means of grouping the separate pieces needed by a program or group of programs. Manipulating a single library file takes less time than if the various pieces it contains were each within an individual file. Libraries generally contain routines relating to a certain area of application; they can be used for functional groupings much as units can. Thus, you might want to maintain a math library, a data file-management library, and so forth—each of these libraries could contain routines general enough to be used by many programs over a long period of time.

Individual programs might also take advantage of the library construct. If a program uses several units suitable for compiling separately, but the units themselves are too small to warrant putting each into its own file, you would want to construct a single library containing all of those units.

Even if a file contains only a single unit or routine, it is treated as a library when the unit or routine is used by some external host.

Library is useful for putting units into SYSTEM.LIBRARY or other libraries and grouping assembly routines together.

This section uses the term compilation unit. A program or unit and all the segments declared inside it are called a compilation unit. The segment for the program or unit is called the host segment of the compilation unit. Segment routines declared inside the host are called subsidiary segments. Units used by the host are not segments belonging to that compilation unit. Units used by the compilation unit generate information in the host segment called segment references. The segment references contain the names of all segments referenced by a compilation unit, and the operating system uses this information to set up a run-time environment.

Some routines called from hosts exist in units in the operating system, and therefore appear in segment references, even though there is no explicit USES declaration. For example, WRITELN resides in the operating system UNIT PASCALIO, so the name PASCALIO appears in the segment references of any host that calls WRITELN.

Using Library

When Library is executed, it displays a prompt asking for an output file name. The file name must end in .CODE. Library removes an old file with the same name as the new library.

Library then displays a prompt asking for the input file name. .CODE is automatically appended.

Library Example

You specify SCREENOPS.CODE as an input file. Library displays the following listing.

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

Input file? SCREENOPS<ret>
 0 u SCREENOP 582
 1 s SEGSCINI 508
 2 s SEGSCPRO 229
 3 s SEGSCCHE 126

Output file? NEW.CODE<ret>
```

The preceding display shows that the file SCREENOPS consists of a unit. There are four possible types of code that can occupy the 16 slots in a library: units, programs, segment routines, and assembled routines. Library displays the type, along with the name and length (in words) of each module.

Library's menu shows the various commands available.

- The N(ew command displays a prompt asking for a new input file.
- The A(bort command stops Library without saving the output file.
- The Q(uit command stops Library and saves the output file. Then Library displays the prompt, `Notice?`, at the top of the display unit. Enter copyright notice and press the **RETURN** key. It is placed in the output file's segment dictionary. Pressing the **RETURN** key without entering a copyright notice exits Library without writing a copyright notice.

- The T(og command toggles a switch that determines whether or not INTERFACE parts of units are copied to the output file.

- The R(efs command lists the names of each entry in the segment reference lists of all segments currently in the output file. The list of names also includes the names of all compilation units currently in the output file, even though their names may not occur in any of the segment references.

The remaining five commands allow code segments to be transferred from the input file to the output file.

- A given slot can be transferred to the output file by typing a digit (zero through nine). Library then displays a prompt: `Copy from slot#?` along with the digit just entered. If that is the name of the slot, press the space bar. If that is the first digit of a two-digit slot number, enter the second digit and press the space bar. Library confirms the entry before actually copying code. Press the **BACKSPACE** key to correct errors. If you press the **RETURN** key without entering a number, the copy does not happen and Library redisplays its menu.

  If the destination slot in the output file is already filled, the system displays a warning and no copy takes place. If an identical code segment is already present anywhere in the output file, the new code segment is copied anyway.

- The E(very command copies all of the codes in the input file to the output file. If, for any code segment, the corresponding slot in the output file is already filled, then Library searches for the next available slot and places the code there. If, for any code segment, an identical code segment already exists in the output file, that segment is not copied over.

- The S(elect command causes Library to display a prompt asking which code segments to transfer. For each code segment not already in the output file, Library displays the prompt: `Copy from slot #__?`. Pressing Y or N causes the segment to be copied or passed by; pressing E causes the remainder of the code segments to be transferred (as in E(very); pressing the space bar or the **RETURN** key aborts the S(elect. If the corresponding slot in the output file is filled, Library searches for the next available slot and places the code there.

- C(omp-unit causes Library to display the prompt: `Copy what compilation unit?`. The compilation unit named is transferred along with any segment procedures that it references. Procedures already present in the output file are not copied.

- F(ill does the equivalent of a C(omp-unit command for all the compilation units referenced by the segment references in the output file.

- I(nput allows you to view the next group of segments in an input file. The Library Utility circularly displays 27 segments at a time. If you have over 27 segments in your input file this allows you to view them.

- O(utput allows you to view the next group of segments, past the initial 27 segments shown, in your output file. The I(nput and O(utput commands allow you to toggle back and forth between groups of segments in your files.

NATIVE CODE GENERATOR

Native code generator (NCG) is a utility program that translates selected portions of an executable p-code file into 8086 native code (n-code) on the Texas Instruments Professional Computer. 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 translates only valid executable code files produced by a UCSD p-System compiler.

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

You indicate that portions of code are to be translated by inserting native code directives into the Pascal source file before compilation. The following compiler options are the native code directives.

$N+ and $N−

You insert the first switch {$N+} where the translation should begin and the last switch {$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−}.

Native Code Directives and Pascal

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

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

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

Running the Native Code Generator

The NCG is run by executing 8086.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 can 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 the **RETURN** key in response to the prompt.

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

```
Final 8086 VERSION NUMBER Code for
segment M       procedure 2 segment offset 32
Source Object                          .RADIX    10
P-Code N-Code                  MP      .EQU      BP
(Dec. Offsets)                 BASE    .EQU      DX

                      0|                        .WORD     27,0
                      0|
4:                    0|  A8          ;p-code   NATIVE
                      1|  8B4604                MOV       AX,4[MP]
                      4|  3B4602                CMP       AX,2[MP]
                      7|  7E08                  JLE       L1
9:                    9|  8B4604                MOV       AX,4[MP]
                     12|  894606                MOV       6[MP],AX
11:                  15|  EB06                  JMP       L2
13:                  17|  8B4602      L1:       MOV       AX,2[MP]
                     20|  894606                MOV       6[MP],AX
15:                  23|  FF1E0400    L2:       CALL      DS:4
15:                  27|              ;exit code
                     27|  9602        ;p-code   RPU       2
```

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

- NATIVE is the p-code that instructs the interpreter to start executing n-code. Execution starts on the byte following the NATIVE instruction.

- The header lists the register conventions: p-machine registers on the left and processor registers on the right.

- Line L3 contains the instruction that returns the processor from n-code to p-code.

- On the 8086, global and external variables are referenced through register DX, which contains Base.

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 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.

Limits of Native Code Generation

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

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

Sequences of straight n-code (code between a NATIVE instruction and its matching return instruction, see individual processor listings) are treated by the p-machine as a single p-code. This fact causes two problems. First, although the **BRK** key may be recognized by the interpreter 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 cannot be terminated by pressing the **BRK** key. Second, p-machine events (interrupts), like the **BRK** key, are only acted upon at p-code boundaries.

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

Some unusual Pascal constructs create code that the NCG will not translate. For example, using the Pascal primitive, p- machine, to generate an RPU instruction results in the error message: Undefined label.

PATCH

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

Patch is meant to be used interactively with a display unit. It uses the screen control module (see *UCSD p-System 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(ixed	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(ixed).
W(ipedisplay	Clears the display of the block off the display unit.
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 display unit-oriented editor, allows the information on the screen to be modified by moving the cursor and entering over previously 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 the CONTROL and C keys simultaneously (the CONTROL-C key combination may be referred to as the ETX key). 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 is 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 interpreter and/or the Basic Input/Output Subsystem [BIOS].)

The width of the output can be controlled; a line can contain any number of machine words: 15 words fill a 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 is not range-checked.

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

D — 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; < = (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 does not matter.

J — Display each word as a decimal integer.

K — Display each word as hexadecimal digits in byte order.

L — Display each word as 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.

U — Pressing Q returns the to EDIT mode; DUMP remembers the current specifications.

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.)

PRINT SPOOLING

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

When SPOOLER is X(ecuted, the following menu appears:

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

The following paragraphs explain the items on this menu:

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

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

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

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

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

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

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

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

If an error occurs (for example, a nonexistent file is specified in the queue), the error message appears only when the p-System is at the command menu. If necessary, the spooler waits until you return to the outer level.

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

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

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

To restart the print spooling process if SPOOLER.CODE is executing when the system goes down, reboot the system, press X(ecute from the command menu, enter *SPOOLER.CODE, and press the RETURN key. Then press R(esume.

REALCONV UTILITY

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

The real constant conversion utility is a filter that works on code files, replacing canonical reals with run-time reals in-place. Hence, when the source file is not available, you should make a backup copy of the code file to be processed before executing the utility program. This avoids destroying the code file while executing REALCONV with an unsuccessful write. Although slight, you should consider this possibility.

Because the conversion algorithm uses real arithmetic of the host processor, the utility must be executed on the processor on which the output file will run. In most cases, a code file produced by the utility will not run on another processor, which reduces the portability of otherwise completely transportable code.

To use the utility, X(ecute REALCONV from the command menu. It responds with the following prompt:

```
Enter filename, <ret> to quit:
```

Respond by entering the name of the code file to be processed, followed by pressing the **RETURN** key. You do not have to append the suffix .CODE.

If REALCONV cannot find the file, it prints the message *File not found* and asks you to enter the file name again. Once a correct file is entered, REALCONV begins translating.

If REALCONV cannot complete the conversion successfully, it prints a message and stops. The messages can be:

```
not enough memory

error in reading...
  The dots stand for:
    segment dictionaries
    first block
    constant pool
    segment
    (as the case may be).

error in writing segment

too many dictionaries
```

Not enough memory means that the segment to be processed is larger than the available memory space.

If the message is *error in reading...*, X(ecute REALCONV again.

If the message is *error in writing segment*, then, before X(ecuting REALCONV again, you have to restore the code file. Restoring the code file depends on the availability of the source file. If the source file is available, compile it again and save the code file. If only the code file was originally available, make a copy of the backup code file. (Remember to backup the original code file.)

Too many dictionaries means that you have more than 80 segments in the file.

The probability of getting any of the three messages is extremely slight, but it can happen.

If REALCONV executes successfully, a dot is written on the console for each segment converted; and, once the conversion is completed, the message *Enter file name:* is displayed so you can process another file. When there are no more files to process, answer the prompt by pressing the RETURN key. This exits REALCONV and returns you to the command menu.

XREF — 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. A table of calls each procedure made along with all nonlocal variable references is also provided. 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 is not the Pascal definition, but it 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 is not 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 is not local to the procedure in question, then the procedure in which the variable was declared is listed.

Variable references are shown in three forms:

- < variable name> ::= a local variable
- < procedure name> < variable name> ::= a variable defined in < procedure> 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

This file contains 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 International Standards Organization (ISO) draft standard.

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

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

- How wide is your output device? [40..132]:

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

- Please enter the file you wish cross-referenced:

  The name of the text file that contains the Pascal program to be referenced. If the specified file cannot be successfully opened, the prompt is repeated until you enter a valid input file name or press the RETURN key. Entering an empty file name, (pressing the RETURN key) 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 the user that the line numbers referenced are synchronized with the line numbers the compiler generates.

- Do you want intrinsics listed? [y/n]:

  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, are not expected to have a BEGIN associated with them.

- Do you want initial procedure nestings? [y/n]:

  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? [y/n]:

  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 the user selects warnings, then that person has 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 the user would like the output directed. If the file is a disk file, the name should have .TEXT appended to it.

The referencer expects to read a complete and syntactically correct Pascal program. Although results with syntactically incorrect programs are not assured, the referencer is not 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 does not 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 is 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 is not 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: the general structure of procedure headings; and 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 does not 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 UCSD Units; it processes each correctly.

A

Special Keys

BACKSPACE	–	Backspace
TAB	–	TAB
UP ARROW	–	
DOWN ARROW	–	
LEFT ARROW	–	
RIGHT ARROW	–	Marked Accordingly
RETURN	–	Return
ETX	–	CTRL C
ESC	–	ESC
BREAK	–	Shifted BRK/PAUSE
STOP	–	Unshifted BRK/PAUSE or CTRL S
FLUSH	–	CTRL F
DELETE LINE	–	CTRL Backspace
EXCHANGE-INSERT	–	INS
EXCHANGE-DELETE	–	DEL
ALPHA-LOCK	–	Uppercase

The following table describes special keyboard functions and the affected keys:

Keys Pressed	Function
ALT 1	Toggle display intensity (color)
ALT 2	Toggle cursor size
ALT 3	Toggle cursor state
ALT 4	Clear graphics screens
ALT 5	Disable function keys

The special function keys provides the following features:

ALT 1 — The display intensity toggle allows you to set the intensity of characters displayed on the gray scale monitor. On the high resolution graphics monitor, this sets the color of the displayed characters.

ALT 2 — The cursor size toggle allows you to define the cursor ranging from no cursor to a full block cursor.

ALT 3 — The cursor state toggle allows you to turn the cursor off, cause the cursor to blink rapidly, cause the cursor to blink slowly, or cause the cursor to be nonblinking.

ALT 4 — The clear graphics display unit function clears the installed graphics planes without affecting the text display unit.

ALT 5 — The disable function keys toggle allows you to enable or disable function keys on the keyboard. The affected keys are those that return two character sequences. These keys are not always appropriate and may be confusing when struck in certain circumstances since the application program will receive two characters which are meaningful only when used as a pair. For example, with function keys enabled, the F1 key causes the assembler to execute when the system is displaying the Command menu. The F1 key returns the characters 27 and 97. The system discards the 27 as invalid and treats the 97 (lowercase a) as a command to begin the assembler. This toggle allows you to disable these two key sequences. The same keys are used to re-enable the function keys when desired.

B

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, Breakpoint
15 Bad Block

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 the user must rebootstrap.

C

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

Device Number Assignments

The p-System on the Texas Instruments Professional Computer supports up to 4 disk drives, a random access memory (RAM) disk, a parallel printer, the serial communications option card, and up to 10 subsidiary volumes. The following table describes the shipped unit number assignments as defined by SYSTEM.MISCINFO.

Device Number	Volume Name	Comment
1	CONSOLE	
2	SYSTERM	
4	DS01	Volume name assigned by user
5	DS02	Volume name assigned by user
6*	PRINTER	Optional printer required
7*	REMIN	Optional communications required
8*	REMOUT	Optional communications required
9*	DS03	Optional disk drive required
10*	DS04	Optional disk drive required
11*	RAM disk	Optional memory required
12		First user subsidiary vol
.		
.		
.		
21		Last user subsidiary vol

* Support for optional devices is included in standard device support. However, device not required for system operation.

E

ASCII Codes

Decimal	Octal	Hexadecimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!

Decimal	Octal	Hexadecimal	Character
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J

Decimal	Octal	Hexadecimal	Character
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	/
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	‘
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s

Decimal	Octal	Hexadecimal	Character
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

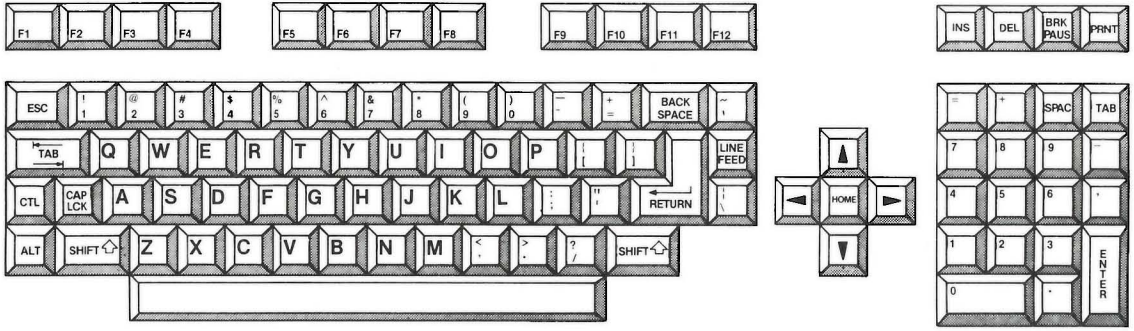
Keyboard Mapping

The following figures describe the Texas Instruments Professional Computer keyboard and key code sequences.

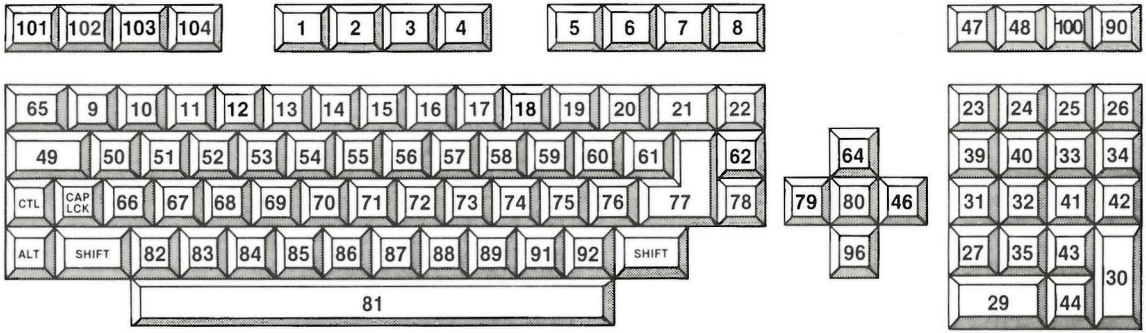
The first figure illustrates the keyboard and shows the characters that appear on the key caps.

The second figure is laid out in the same fashion and indicates the key numbers for each of the keys in the first figure.

The table at the end lists the keys by key number and indicates what codes they produce when entered normally or in conjunction with CTL, ALT, and SHIFT.


F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
F11
F12
INS
DEL
BRK PAUS
PRNT
ESC
BACK SPACE
TAB
LINE FEED
CTL
CAP LCK
RETURN
ALT
SHIFT
SHIFT
HOME
SPAC
TAB
ENTER

2264071


101
102
103
104
1
2
3
4
5
6
7
8
47
48
100
90
65
9
10
11
12
13
14
15
16
17
18
19
20
21
22
49
50
51
52
53
54
55
56
57
58
59
60
61
62
CTL
CAP LCK
66
67
68
69
70
71
72
73
74
75
76
77
78
ALT
SHIFT
82
83
84
85
86
87
88
89
91
92
SHIFT
81
64
79
80
46
96
23
24
25
26
39
40
33
34
31
32
41
42
27
35
43
30
29
44

2284070

Key	Normal			Shift			Control			Alt		
01	F5	101	*		88	*		98	*	108	*
02	F6	102	*		89	*		99	*	109	*
03	F7	103	*		90	*		100	*	110	*
04	F8	104	*		91	*		101	*	111	*
05	F9	105	*		92	*		102	*	112	*
06	F10	106	*		93	*		103	*	113	*
07	F11	108	*		08	*		10	*	12	*
08	F12	110	*		09	*		11	*	13	*
09	1	49		!	33			--		--	
10	2	50		@	69			03	*	--	
11	3	51		#	35			--		--	
12	4	52		$	36			--		--	
13	5	53		%	37			--		--	
14	6	54			94		RS	30		125	
15	7	55		&	38			--		126	*
16	8	56		*	42			--		127	*
17	9	57		(	40			--		128	*
18	0	48		)	41			--		129	*
19	-	45		-	95		US	31		--	*
20	=	61		+	43			--		--	*
21	BS	08		BS	08		DEL	127		--	*
22	`	96		~	126			--		--	*
23	=	61		=	61		=	61		140	*
24	+	43		+	43		+	43		141	*
25	SP	32		SP	32		SP	32		142	*
26	HT	09		Bktab	15	*	HT	09		143	*
27	1	49		1	49		1	49		--	
28		--			--			--		--	
29	0	48		0	48		0	48		--	
30	CR	13		CR	13		CR	13		--	
31	4	52		4	52		4	52		--	
32	5	53		5	53		5	53		--	
33	9	57		9	57		9	57		--	
34	-	45		-	45		-	45		--	
35	2	50		2	50		2	50		--	

Key	Normal			Shift			Control			Alt		
36		--			--			--			--	
37		--			--			--			--	
38		--			--			--			--	
39	7	55		7	55		7	55			--	
40	8	56		8	56		8	56			--	
41	6	54		6	54		6	54			--	
42	,	44		,	44		,	44			--	
43	3	51		3	51		3	51			--	
44	.	46		.	46		.	46			--	
45		--			--			--			--	
46	C-rt	67	*		138	*		116	*		78	*
47	Ins	111	*		40	*		41	*		42	*
48	Del	112	*		56	*		57	*		58	*
49	HT	09		BKTAB	15		HT	09			--	
50	q	113		Q	81		DC1	17			16	*
51	w	119		W	87		ETB	23			17	*
52	e	101		E	69		ENQ	05			18	*
53	r	114		R	82		DC2	18			19	*
54	t	116		T	84		DC4	20			20	*
55	y	121		Y	89		EM	25			21	*
56	u	117		U	89		NAK	21			22	*
57	i	105		I	73		HT	09			23	*
58	o	111		O	79		SI	15			24	*
59	p	112		P	80		DLE	16			25	*
60	|	91		|	123		ESC	27		--	*
61	|	93		|	125		GS	29		--	*
62	LF	10		LF	10			117	*		79	*
63		--			--			--			--	
64	C-up	65	*		136	*		132	*		73	*
65	ESC	27		ESC	27		ESC	27			--	
66	a	97		A	65		SOH	01			30	*
67	s	115	*	S	83		DC3	19			31	*
68	d	100	*	D	68		EOT	04			32	*
69	f	102		F	70		ACK	06			33	*
70	g	103		G	71		BEL	07			34	*

Key	Normal			Shift			Control			Alt		
71	h	104		H	72		BS	08			35	*
72	j	106		J	74		LF	10			36	*
73	k	107		K	75		VT	11			37	*
74	l	108		L	76		FF	12			38	*
75	;	59		:	58			--			--	
76	`	39		"	34			--			--	
77	CR	13		CR	13		CR	13			--	
78	\	92		¦	124		FS	28			--	
79	c-lf	08	*		139	*		115	*		76	*
80	Home	107	*		134	*		119	*		133	*
81	SP	32		SP	32		SP	32		SP	32	*
82	z	122		Z	90		SUB	26			44	*
83	x	120		X	88		CAN	24			45	*
84	c	99		C	67		ETX	03			46	*
85	v	118		V	86		SYN	22			47	*
86	b	98		B	66		STX	02			48	*
87	n	110		N	78		SO	14			49	*
88	m	109		M	77		CR	13			50	*
89	,	44		<	60			--			--	
90	Print	114	*		--			--			--	
91	.	46		>	62			--			--	
92	/	47		?	63			--			--	
93		--			--			--			--	
94		--			--			--			--	
95		--			--			--			--	
96	C-dn	66	*		137	*		118	*		81	*
97		--			--			--			--	
98		--			--			--			--	
99		--			--			--			--	
100	Ppau	**		Pbrk	128			--			--	
101	F1	59	*		84	*		94	*		104	*
102	F2	60	*		85	*		95	*		105	*
103	F3	61	*		86	*		96	*		106	*
104	F4	62	*		87	*		97	*		107	*

* - Preceded by keyboard lead-in

-- - Key not used, no key code returned to user program

Pascal Compiler 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 a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Respecified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: A forward declared function's result type 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: Standard file was redeclared
172: Undeclared external file
173: FORTRAN procedure or function expected
174: Pascal function 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

Pascal 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 7 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.

Error Number	Comments
6	The operator (variable, constant, jump location) that the back-end is trying to remap is not in the scope of the compilation unit.
7	The back-end cannot 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 zero or greater than the length of the intermediate code file while building a jump table.
12	A jump site cannot 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 cannot be found in the jump table.
25	The code pointer is less than zero or greater than the length of the intermediate code file while generating p-code.
27	The code pointer is less than zero before trying to read in more code from the intermediate code file to the code buffer.

Error Number	Comments
28	The code pointer is less than zero after trying to read in more code from the intermediate code file to the code buffer.
29	The current final output block number is greater than the block number of the intermediate code file being processed.
30	The final code file size exceeds the intermediate code file size before trying to write more final code.
31	The final code file size exceeds the intermediate code file size after writing more final code.
41	The line length of a compiled listing exceeds 120 characters. (Note: This error can occur on a pre-IV.1 compiler if there is an illegal character after a DLE character.)
86	Could not find a particular segment in the intermediate code file.
99	The number of procedures does not match the number specified in the procedure dictionary.

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

Title	Page
A	
A(bort	6-14
Altering memory	5-7
ANSI	3-6
Assembly language	2-28
B	
Breakpoints	5-5
C	
Call structure table	6-33
CHAIN	3-4
Chaining programs	3-4
Code segment	2-27
Command I/O unit	3-17
C(omp-unit	6-16
C(ompile	2-3
Compiled listing	2-7
Compiler	2-3
Compiler options	2-11
$B Begin Conditional Comp	2-12, 2-19, 2-23
$B Conditional Comp Flag	2-20
$B End Conditional Comp	2-23
$C Copyright Field	2-13
$D Conditional Comp Flag	2-13, 2-19
$E End Conditional Comp	2-13, 2-19
$I Include File	2-14
$I I/O Check	2-13
$L Compiled Listing	2-15
$N Native Code Generation	2-16
$P Page	2-16
$Q Quiet	2-17

Title	Page
$R Range Checking	2-17
$R Real Size Directive	2-17
$T Title	2-18
$U Use Library	2-18
$U User Program	2-19
Conditional compilation	2-19
D	
Date_Test	4-42
D_Change_Name	4-35
D_Choice	4-14
D_Code	4-14
D_Data	4-14
D(ebug	
Breakpoints	5-5
Variables	5-6
Debugger	5-3
Decode	6-3
D_Free	4-14
Directories	4-3, 4-10
Directory information	4-10
File type selection	4-13
Notation and terminology	4-11
Directory information access	4-10
Directory lister program	4-28
Directory manipulation	4-11
DIR.INFO	4-10
DIR_INFO	4-10
File type selection	4-13
Notation and terminology	4-11
Disassembling	5-10
Displaying memory	5-7
D_NAME	4-17
D_NameType	4-14
D_Scan_Title	4-13, 4-17
D_SELECT	4-13
D_SVol	4-14
D_Temp	4-14
D_Text	4-14

Title Page

D_TITLE 4-17
D_TYPE 4-18
D_Vol 4-14
D_VOLUME 4-18

E
Error handler unit 3-14
Error handling 4-11
Error results 4-15
E(very 6-15
Extended Backus-Naur Form (EBNF) 4-11

F
File dates 4-15
FILE.INFO 4-4, 4-64
FILE_INFO 4-4, 4-9
File information 4-64
File management units 4-3
  DIR.INFO 4-3
  FILE.INFO 4-4
  SYS.INFO 4-4
  WILD 4-4
File Manipulation 4-11
F(ill 6-16
Function
  Aspect_ratio 3-30
  Command I/O: Redirect 3-18
  Create_Figure 3-31
  D_Change_Date 4-40
  D_Change_Name 4-32
  DIR_INFO: D_DIR_List 4-20
  DIR_INFO: D_Dismount 4-17
  DIR_INFO: D_Krunch 4-16
  DIR_INFO: D_Mount 4-17
  DIR_INFO: D_Scan_Title 4-17
  DIR_INFO: Result 4-28
  D_Rem_Files 4-43
  D_Wild_Match 4-53
  F_is_Blocked 4-65

F_Length 4-64
F_Open 4-64
F_Start 4-65
F_Unit_Number 4-64
Load_figure 3-36
Read_figure_file 3-35
Read_pixel 3-34
SC_Check_Char 3-11
SC_Find_X 3-9
SC_Find_Y 3-9
SC_Has_Key 3-12
SC_Map_CRT_Command 3-11
SC_Prompt 3-10
SC_Scrn_Has 3-12
SC_Space_Wait 3-10
SI_Sys_Unit 4-57
Store_figure 3-36
Turtle_angle 3-26
Turtle_x 3-26
Turtle_y 3-26
Write_figure_file 3-35

I
Implementation section 2-29
Interface section 2-29

L
$L 2-4
Lexical structure table 6-32
Libraries 2-29
Library text file 2-30
Library utility 6-12
Library's menu 6-14
L(inker 2-28
Linking 2-27
Listing 2-7
Locktest 4-46

Title Page

M
Mark stack chain 5-9
Memory 5-7
Meta-words 4-11
Multi-tasking support 4-11

N
N(ew 6-14

O
Operating system user manual 4-12

P
Pascal 1-3
Patch 6-21
  DUMP 6-22, 6-25, 6-33
  EDIT 6-22
  Prompts 6-27
  TYPE 6-23
p-code 5-9, 5-10
Procedure
  Activate_turtle 3-24
  Background 3-28
  Command I/O: Chain 3-18
  Command I/O: Exception 3-19
  Delete_figure 3-32
  Display_scale 3-29
  D_Lock 4-46
  D_Release 4-47
  F_Date 4-66
  F_File_Title 4-65
  Fillscreen 3-28
  F_Volume 4-65
  Get_Figure 3-32
  Move 3-24
  Moveto 3-24
  Pen_color 3-25
  Pen_Mode 3-26
  Put_Figure 3-33

Title	Page
SC_Clr_Cur_Line	3-8
SC_Clr_Line	3-8
SC_Clr_Screen	3-8
SC_Down	3-8
SC_Eras_EOS	3-8
SC_Erase_to_EOL	3-8
SC_GetC_CH	3-10
SC_Goto_XY	3-9
SC_Home	3-9
SC_Init	3-8
SC_Left	3-9
SC_Right	3-9
SC_Up	3-9
SC_Use_Info	3-12
SC_Use_Port	3-13
Set_Error_Line	3-16
Set_pixel	3-35
Set_User_Message	3-16
SI_Code_Tid	4-57
SI_Code_Vid	4-57
SI_Get_Date	4-58
SI_Get_Pref_Vol	4-58
SI_Get_Sys_Vol	4-58
SI_Set_Date	4-58
SI_Set_Pref_Vol	4-58
SI_Text_Tid	4-57
SI_Text_Vid	4-57
Turn	3-24
Turnto	3-25
Viewport	3-34
WChar	3-28
WString	3-28
Procedure call table	6-34
Program	
Date_Test	4-42
D_Change_Name	4-33
Directory lister program	4-28
Locktest	4-47

Rem_Test 4-45
Scan_Test 4-19
Sys_Test 4-59
WildChng 4-36
Wild_Test 4-56

Q
Q(uit 6-14

R
REALCONV 6-29
Referencer's output
Call structure table 6-33
Lexical structure table 6-32, 6-43
Procedure call table 6-34
Variable call table 6-34
Variable reference table 6-34
Warnings file 6-35
R(efs 6-15
R(egister 5-9
Rem_Test 4-45
R(un 2-3

S
Scan_Test 4-19
Screen control unit 3-6
SCREENOPS.CODE 3-6
Segmenting a program 2-27
Segments 2-27, 2-32
S(elect 6-16
Selective uses 2-22
Separate compilation 2-28
External compilation 2-28
Separate compilations 6-12
Single step 5-9
Special Keys A-1
Symbolic debugging 5-12
Syntax errors 2-6

Title Page

SYS.INFO 4-4, 4-57
SYS_INFO 4-8
System Information 4-57
SYSTEM.LIBRARY 2-30
SYSTEM.MENU 3-4
SYSTEM.STARTUP 3-4
Sys_Test 4-59

T
Text files 5-8
T(og 6-15

U
UCSD Pascal 2-3
Unit interface
  DIR_INFO 4-6
  FILE_INFO 4-9
  SYS_INFO 4-8
  WILD 4-5
UNIT PASCALIO 6-13
Units 2-27, 2-28, 2-29, 2-32, 6-4
  Implementation section 2-29
  Interface section 2-29
Use 2-31
User interface 3-3
USERLIB.TEXT 2-30
Using Library 6-13
Using Referencer 6-36

V
Variable call table 6-34
Variable reference table 6-34
Variables 5-6
Vector arrows 6-25

W
Warnings file 6-35
WILD 4-49

Title	Page
Wild Cards	4-49
WildChng	4-36
Wild_Test	4-56
WRITELN	6-13
X	
XREF	6-31

THREE-MONTH LIMITED WARRANTY TEXAS INSTRUMENTS PROFESSIONAL COMPUTER SOFTWARE MEDIA

TEXAS INSTRUMENTS INCORPORATED EXTENDS THIS CONSUMER WARRANTY ONLY TO THE ORIGINAL CONSUMER/PURCHASER.

WARRANTY DURATION

The media is warranted for a period of three (3) months from the date of original purchase by the consumer.

Some states do not allow the exclusion or limitation of incidental or consequential damages or limitations on how long an implied warranty lasts, so the above limitations or exclusions may not apply to you.

WARRANTY COVERAGE

This limited warranty covers the cassette or diskette (media) on which the computer program is furnished. It does not extend to the program contained on the media or the accompanying book materials (collectively the Program). The media is warranted against defects in material or workmanship. THIS WARRANTY IS VOID IF THE MEDIA HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.

PERFORMANCE BY TI UNDER WARRANTY

During the above three-month warranty period, defective media will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below or an authorized Texas Instruments Professional Computer Dealer with a copy of the purchase receipt. The replacement media will be warranted for three months from date of replacement. Other than the postage requirement (where allowed by state law), no charge will be made for the replacement. TI strongly recommends that you insure the media for value prior to mailing.

WARRANTY AND CONSEQUENTIAL DAMAGES DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER ARISING OUT OF THE PURCHASE OR USE OF THE MEDIA. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents:

Texas Instruments
Service Facility
P.O. Box 1444, MS 7758
Houston, Texas 77001

Canadian Residents:

Geophysical Service Inc.
41 Shelley Road
Richmond Hill, Ontario
Canada L4C 5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments
Consumer Service
831 South Douglas St.
Suite 119
El Segundo, California 90245
(213) 973-2591

Texas Instruments
Consumer Service
6700 S.W. 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAM

The following should be read and understood before using the software media and Program.

TI does not warrant that the Program will be free from error or will meet the specific requirements of the purchaser/user. The purchaser/user assumes complete responsibility for any decision made or actions taken based on information obtained using the Program. Any statements made concerning the utility of the Program are not to be construed as expressed or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAM AND MAKES ALL PROGRAMS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAM. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS. THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE PROGRAM. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE PURCHASER/USER OF THE PROGRAM.

COPYRIGHT

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Program Development
2232399-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

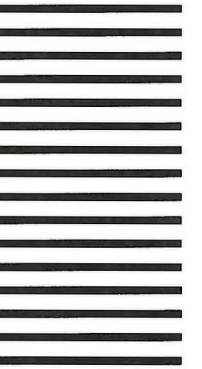
Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Program Development
2232399-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

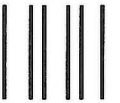
City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Program Development
2232399-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

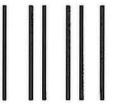
City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

Texas Instruments reserves the right to change its product and service offerings at any time without notice.

TEXAS INSTRUMENTS

2232399-0001

Printed in U.S.A.