Pascal-2™

Version 2.1 for RSX-11

User Manual

October 1986

OREGON SOFTWARE

Name of Contractor and Address:

Oregon Software, Inc.
6915 S.W. Macadam Avenue
Portland, Oregon 97219
Phone: 503-245-2202

ISBN 0-92595801-8

Contents

Preface

This copy of the *Pascal-2 User Manual* for the RSXoperating system corresponds to Version 2.1E of the Pascal-2 software. This edition provides users with an up-to-date, technically accurate manual. We've corrected many technical errors and revised much of the text for clarity. The next edition will provide further improvements, such as a complete description of the Pascal language as implemented.

Many changes resulted from user comments, which we continue to encourage. For example, users' feedback helped us clarify our explanations of the role of the taskbuilder and the use of structured constants. If you wish to contribute the next edition, please fill out and return the evaluation report provided at the end of this manual, or address your comments directly to:

> David Spencer, Manager
> Technical Publications
> Oregon Software
> 6915 SW Macadam
> Portland, Oregon 97219-2397

We appreciate hearing from you.

For our technical publishing, we use TEX, a computer-based typesetting program developed by Don Knuth at Stanford University. This manual is typeset in the Computer Modern Roman family of type faces with the TEX system. Masters were produced at Oregon Software on a Imprint-10 laser printer driven by TEX-in-Pascal and our VAX-11/780. This edition is the last one to be produced on a laser printer. With the next release, masters for the user manual will be produced on a phototypesetter.

Pascal-2 V2.1/RSX Introduction

Pascal-2 is an integrated system for software development. At the heart of the system is a transportable multipass compiler that adheres to the Pascal standard while performing optimizations to generate compact, fast code. The Pascal-2 system also offers sophisticated error checking during compilations, extensive error reporting and recovery at run-time, a Debugger to examine the dynamic state of a running program in a high-level Pascal context, plus other development utilities. Together, these components offer the professional programmer a structured and unified environment in which to design, code, test, maintain, and improve software. Within this environment, more reliable programs may be produced in less time. Further, programs produced with the Pascal-2 compiler are more portable than those produced with many other programming packages.

The Pascal-2 Software Development System

Pascal-2 compiler, the support library, the formatters PASMAT and PB, the Debugger and Profiler, and the cross-references XREF and PROCREF. The text formatter PROSE, and the installation verifier VERIFY, not shown, are also components of the system. The user creates the Pascal source program, the included source files, user libraries, and resident libraries. The Text Editor and Task Builder are supplied by the computer vendor.

Certification

Programs compiled under Pascal–2 are portable and reliable because of the compiler's strict implementation of the standard. Also, Pascal–2 is very consistent from system to system and all Pascal–2 implementations are consistent with each other. Other compilers may differ according to the processor and operating system you're using or may not be available on larger systems.

For Pascal compilers, certification is relatively new. The Federal Software Management Support Center (FSMSC), a private company under contract with the GSA, tests Pascal compilers according to a validation process that measures a compiler's performance on a suite of 740 test programs. The tests subject the vendor's claims to comparison with the published standard. FSMSC testing guarantees that the software meets ANSI (American National Standards Institute) standards and FIPS (Federal Information Processing Society) requirements. FSMSC certification is required for any compiler purchased by an agency of the federal government.

This version of the Pascal–2 compiler has been certified by the British Standards Institution (BSI) and the FSMSC for compliance with ISO Standard 7185, Level 1. During the certification process, Oregon Software submitted the compiler to a test program which automatically generates a report on the software's performance. The release notes contain a copy of this report.

Pascal–2 Documentation Package

The Pascal–2 user documentation contains information on the use of the Pascal–2 compiler and related utilities on Digital's RSX operating systems: RSX-11M, RSX-11M-Plus, IAS, and VAX/VMS-AME. In general, we assume that readers of the manual are programmers familiar with Pascal and the RSX operating system. Some sections assume a detailed working knowledge of the language.

The manual consists of five major guides, as follows:

- The User Guide serves as a quick overview of the Pascal–2 system, to give you a feel for how it works. Written for new users, the guide takes you through the basic steps of compiling, correcting, and running a Pascal–2 program. The User Guide also provides brief explanations and examples of some of the standard features and utilities of the Pascal–2 system.
- The Programmer Reference contains detailed descriptions of compilation commands, embedded and low-level switches, and the low-level interface between Pascal–2 and the operating system. The Programmer Reference also contains a miscellany of information on implementation-related problems, divided into two broad categories: error recovery and implementation notes. Finally, the reference describes Pascal–2's optimizations and provides helpful hints as to the cause of compile-time and run-time errors and ways to fix the errors.

- The Language Specification describes Pascal–2's language features in detail. Since the second edition of Jensen and Wirth's *User Manual and Report* in 1978, the language has undergone major changes, which are incorporated in the international Pascal standard, ISO 7185. Because not everyone is familiar with that document, the Language Specification begins by summarizing those changes and describing the ways that Pascal–2 implements them. Thus, the guide serves not only as a description of our Pascal product but also as a review of the language's evolution since 1978.

- The Debugger and Profiler Guide describes two programs designed to alleviate tedious aspects of programming and to improve the usefulness of the Pascal–2 system. The Debugger helps find and correct errors that cannot be caught at compile time. The execution Profiler shows less efficient areas of the program in terms of the number of statements executed.

- The Utilities Guide describes each of the following packages: program formatters, a text formatter, cross-reference programs, a package that helps interface assembler routines with Pascal–2 programs, and a dynamic string package. Each utility is described in detail, with examples.

A set of system-specific release notes accompanies each shipment of the software. These release notes contain installation procedures and inform the user of software bugs and documentation errors. The release notes should be kept with the *Pascal–2 User Manual* for reference.

The *Pascal–2 User Manual* is not intended to be a Pascal textbook. Beginners can make their way carefully through this manual, but we refer you to the reading list in the appendix, "For More Information."

For information on the RSX system, see these RSX manuals: *Introduction to RSX*, *MCR Operations*, *I/O Operations Reference Manual*, *Task Builder*, *Executive Reference*.

In addition, Pascal–1 customers upgrading to Pascal–2 should refer to the *Pascal–2 Conversion Guide* and the CONVRS utility, which are available from Oregon Software. The *Conversion Guide* explains specific language differences between Pascal–1 and Pascal–2 and the practical programming problems created by the differences. The guide describes the use of the CONVRS utility to help isolate areas in a Pascal–1 program that will have to be modified to convert to Pascal–2; the guide then details the steps required to convert the programs. The *Conversion Guide* concludes with a list of solutions to errors that you may encounter while completing the conversion to Pascal–2.

Style Notes

The *Pascal-2 User Manual* follows these style conventions:

Text: Pascal reserved words, predefined symbols, switches and compiler directives are in boldface typewriter type: **`begin`**, **`write`**, **`%include`**, **`nomain`**. Portions of examples referred to in text are in boldface typewriter type. System directives are in upper-case boldface typewriter type: **`WAITFR`**, **`SPAWN`**. Program and system names are in upper case: ROTAT, RSX.

Program Examples: Commands that you should type are in underlined boldface typewriter: RUN EX. These commands assume a carriage return at the end.

Program Listings: The Pascal-2 compiler accepts any combination of upper-case and lower-case characters. Examples in this manual have Pascal words in lower case and have user-defined words with an initial capital letter and other capitalization as needed for readability, as shown in this program segment:

```
procedure Show;
begin
  SomeUserAction;
  writeln(Result);
end;
```

Single quotes ('..') in examples and in text appear as "..".

Terminology: We use standard terms as they are used in documents describing the RSX operating system.

Pascal–2 V2.1/RSX User Guide

This introductory section gives you a tutorial overview of Pascal–2's features. The User Guide explains how to compile and run Pascal–2 programs, how to interpret program listings and error messages, and how to use some of the utility programs. This section also provides a complete list of switches used to control the compilation process.

This guide assumes that you are familiar with: simple RSX commands, a text editor (e.g., EDIT, TECO, EDT, SOS), and elementary Pascal programming.

This section does not provide:

- An introduction to Pascal (see *Programming in Pascal* by Peter Grogono);
- A detailed description of Pascal–2 (see the Language Specification, and Doug Cooper's *Standard Pascal User Reference Manual*);
- An expert's guide to Pascal–2 (see the Programmer Reference).

Getting Started

The first step in running a Pascal program is to enter the program into the computer and store it in the file system. Use a familiar text editor to enter the program; store it in a file with the file type .PAS. The Pascal–2 compiler accepts free-format program files, so use blanks, tabs, new lines, and form feeds as desired to help make the program readable.

This Pascal version of a program is called the source program, or the source file. All other versions of the program are translations from the source program.

Compiling the Program

After editing, you must compile your program—translate it into a form that the computer can execute. The Pascal-2 compilation process is directed by the PAS system task. The PAS command causes the Pascal-2 compiler to produce an .OBJ object file. The Task Builder combines the object file with the Pascal-2 library to produce an executable task image. With a source file called TEST.PAS, the entire compilation process follows this example:

```
>PAS TEST
>TKB TEST/FP/CP = TEST,LB:[1,1]PASLIB/LB
```

As the example shows, the .PAS, .TSK, and .OBJ extensions may be omitted from the file names on commands to the Pascal-2 compiler and the Task Builder. These extensions must, however, be included in commands to other RSX system programs such as the editor.

This example shows how you may compile a single program. Subsequent examples show batch mode and command-file compilations.

Notice, too, the /FP and /CP switches in the Task Builder command. The /FP switch enables the saving of floating-point context, and the /CP switch enables checkpointing. We strongly recommend these switches for Pascal tasks. See the "Compilation Options" for a complete list of switches.

To illustrate the compilation process, let's say that the program

```
program First (output);
begin
  write ('"Things are best in their beginnings"');
  writeln (' -- Blaise Pascal');
end.
```

is stored in the file FIRST.PAS.

Compilation proceeds as follows:

```
>PAS FIRST
>TKB FIRST/FP/CP = FIRST,LB:[1,1]PASLIB/LB
>RUN FIRST
"Things are best in their beginnings" -- Blaise Pascal
```

Notice that no errors were detected. The next example shows what happens if detectable errors are present in the source program.

Checking For Errors

The Pascal-2 compiler detects nearly 150 types of "grammatical" errors in a program: errors in syntax such as missing semicolons, undefined identifiers, missing **begin** and **end** reserved words, and similar mistakes. As an example, the following program contains a deliberate error: a semicolon is missing between the program heading and the reserved word **begin**.

```
program Second (output)
begin
  writeln ('Things get worse as they continue');
end.
```

Semicolon errors (the most common errors made by beginning Pascal programmers) are always detected by the compiler:

Example: Syntax Error Message on Screen

```
>PAS SECOND

Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM    Site #1-1    Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 SECOND

    1      program Second (output)
                                   ^19
*** 19: Use ';' to separate statements


*** There was 1 line with errors detected ***
?Errors detected: 1
```

For each detected error, a line of the source program is printed, then an arrow indicating the approximate position of the error, then a message describing the error. (The number "19" is the error message number generated by the compiler.) See Appendix A of the Programmer Reference for a complete list of detectable compilation errors.

The Program Listing

Many times, to correct an error, you need to see more of the program than just the line on which the error appears. The Pascal-2 compiler can be directed to display the entire program, with all detected errors and other information. This is the "listing" of the program.

To obtain a listing file (.LST), include the **list** switch in the compilation command line:

```
>PAS SECOND/LIST
```

To get a program listing at a terminal, specify `TI:` as the listing file, as shown below. The listing also may be written to the line printer or a disk file.

Example: Program Listing With Compilation Error Message

```
>PAS THIRD,TI:= THIRD/LIST

Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM     Site #1-1     Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 THIRD,TI: = THIRD/LIST

        1       program Third (output)
                                            ^19
*** 19: Use ';' to separate statements
        2       begin
        3         writeln ('Love or hate alters the aspect of justice');
        4       end.

***There was 1 line with errors detected ***
```

The listing is printed in pages, with a heading on each page showing the program name, the exact version of the Pascal-2 compiler, the date and time, and the licensed user identification. The listing also prints out, in the left-hand column, the line number for each line of the program. You also may use the `errors` switch to create a listing file containing only the lines with detected errors.

As illustrated in the example of `list`, a compilation switch modifies the compilation process in some way. A switch is signified by a slash and a descriptive name. All of the compilation switches are explained in the "Compilation Options" section of this guide.

Compiler Commands

All Pascal-2 compilation commands are divided into three parts: the compiler invocation command, the file specifications, and the compilation switches.

The compilation syntax for Pascal-2 is this:

>PAS *output-file, listing-file=input-files/switches*

The **PAS** invocation (or some other name that your system manager has chosen for the invocation command) must always come first; it may be written in any combination of upper-case or lower-case characters. **PAS** may be followed on the same line by the rest of the compilation specification or entered on a line by itself. In the latter case, the **PAS>** prompt appears for the file specifications and compilation switches.

input-files: The only required file specification is at least one input file. Multiple input files are concatenated in order, from left to right, so that a large program can be split into separate files or so that a common set of definitions can be placed in a configuration file. With "source concatenation" no input file can contain a **program** statement, except for the first file listed. If no output specification is given, the output is determined by the compilation switches; the file name is taken from the last input file specified; and the output files will be placed in the default directory. The default input file extension is .PAS. Multiple input files are separated by a comma.

output-file: The output file specifies the name of the object output, with a default extension of .OBJ. If the **macro** compilation switch is specified, the output file contains MACRO-11 code and the default extension is .MAC.

listing-file: The listing file specifies the file to receive the compilation or error listing. The default listing file extension is .LST.

If an equal sign appears on the command line, but no file name is listed in the position of the output file, no output file is generated. If no file name is listed in the position of the listing file, a listing output is produced only if errors exist; if errors exist, output is sent to the user's terminal with the **errors** switch assumed.

switches: Program compilation is affected in some way by one or more of the options described in the next section. Examples in this manual show the compilation switches after the last file specification, but switches may appear after any file specification and wherever they're placed, they apply to the entire compilation. Multiple switches are separated by slashes.

You can also run the compiler directly, responding to the prompt as shown here:

```
>RUN [1,54]PASCAL
PAS>  ———————— prompts for the rest of the command line
```

Compilation Options

The Pascal–2 compiler provides you with a number of options, which are implemented by switches entered on the command line or embedded in the code itself. Switches may be used to change the compiler's characteristics or to include a utility program.

Compilation Switches

Compilation switches provide control over the files generated and over some aspects of the generated code. A switch is signified by a descriptive name (e.g., `check`). A switch name beginning with `no` reverses the effect of the switch (e.g., `nocheck`). A switch name may be abbreviated as long as the shortened form is sufficient to identify the switch. Three characters of the switch name (excluding the `no`) always identify a Pascal–2 compilation switch (e.g., `che`, `noche`; `mac`, `nomac`).

Some switches, such as `object` and `macro`, are incompatible, causing the error message "conflicting switches specified" if used in the same compilation.

Pascal–2 compilation switches are:

Program Options

`double` Specifies that all real arithmetic is to be done with double-precision rather than with single-precision. All `real` variables are in 8-byte floating-point format. You must use colon notation (e.g., `X:18:15`) within the program to obtain double-precision values in the `write` statement. Default is "off": with `real` variables are in 4-byte format. See "Extended Precision" in the *Programmer's Reference* for more details.

`pascal1` Specifies that the interface with external procedures is compatible with Pascal–1. This interface is less efficient than Pascal–2's and may slow program execution. The `pascal1` switch simplifies the conversion of programs from Pascal–1 to Pascal–2 and should be used only when required. Default is the Pascal–2 interface.

`nomain` No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is `main`: a main program is being compiled.

`own` Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with `own`. Default is "off": global variables are shared. With this switch, you can distinguish between "public global" and "private global" variables.

`nowalkback` Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. The `debug` switch disables the generation of the walkback. Default is `walkback`: the tables are generated, and the full walkback in source terms is displayed after the run-time error message. See "Run-Time Error Reporting" later in this section for a discussion of the error walkback. The `walkback` switch in external modules must be set to match those of the main program segment.

Compiler Options

`errors` Requests that the listing file contain only lines with errors. By default, this "errors only" listing is printed on the terminal. You can direct the listing to a disk file by providing a listing-file specification on the command line. The `errors` switch is incompatible with the `list` switch. It also has no effect when used with the `debug` switch or the `profile` switch, because both of these switches always generate a listing file.

`list` Requests a full source listing in the listing file. If a listing file is specified, `list` is assumed unless you specify the `errors` switch. If no listing file is specified, `list` directs output to a listing file with the same name as the first input file and the appropriate extension.

`debug` Requests generation of code and auxiliary files to interface with the Pascal-2 Debugger. Default is "off." The `debug` switch disables the generation of the walkback and affects some optimizations (see "Compiler Optimizations"). This switch cannot be used with the `profile` switch or the `errors` switch.

`profile` Requests an execution profile when the program is run. Default is "off." The switch cannot be used with the `debug` switch or the `errors` switch.

Code Switches

`object` Generates an object format output file with default extension .OBJ. Default is normally "on"; object code is generated. The switch is "off" when `noobject` is specified or when no output file is provided on the command line. The switch cannot be used with the `macro` switch.

`macro` Generates MACRO-11 code in the output file. This code may be assembled by the `MACRO` assembler command to produce an object file. When `macro` is specified, `object` is set "off" and the default extension for the output file becomes .MAC. Default of `macro` is "off." The `macro` switch cannot be used with the `object` switch.

Checking Switches

`nocheck` Disables all run-time checks, including index range checks, subrange assignment checks, pointer checks, stack checks, case label checks, and divide-by-zero checks. Note that compilation errors are still detected. Thus, if `nocheck` is specified, `var A:array [2..10] of integer; A[1] := 0;` is still detected as a compilation error, but `I := 1; A[I] := 0;` is not. After a program has been fully debugged, the `nocheck` switch may be used to reduce the size of the compiled code. Default is `check`.

`standard` Requests that all Pascal-2 extended language features be flagged as errors. Default is `nostandard`.

`times` Prints wall-clock time consumed by the compiler and the compilation rate in lines per minute. Default is "off."

Processor Switches

The processor switch defaults to the processor option for the machine on which the compiler is running. Change the value by specifying one of these four switches on the command line:

`fpp` Requests the compiler to generate code for a machine with the Floating Point Processor (FPP) option. FPP instructions include `ADDF`, `MODF`, `DIVF`, etc. This switch implies the `eis` switch and may not be specified at the same time as the `fis` switch.

`fis` Requests the compiler to generate code for a machine with the Floating Instruction Set (FIS) option. FIS supports only the four basic floating-point instructions and is available on only a few types of machines. This switch implies the `eis` switch and may not be specified at the same time as the `fpp` switch.

`eis` Requests the compiler to generate code for a machine with the Extended Instruction Set (EIS) option. The EIS processor option includes instructions to perform integer multiplication and division. Floating-point operations are done with calls to a floating-point simulator.

`sim` Requests code with calls to software routines for integer multiply and divide as well as for floating-point arithmetic. Should be used only if the target machine does not have EIS.

Embedded Switches

Some characteristics of the compiled code may be controlled by switches included in the source code. These switches take the form of a Pascal comment beginning with a dollar sign '`$`' and followed by a descriptive name, for example:

```
{$indexcheck}
```

A switch name beginning with "no" reverses the effect of the switch, for example:

```
{$noindexcheck}
```

Most switches may be abbreviated to a minimum of three characters, for example:

{**$ind**} or {**$noi**}

However, when using **$nopointercheck** and **$noprofile** be sure to enter more than three characters, or the compiler treats the switch as an ordinary comment.

Multiple switches may be embedded within a single comment. The switches must be separated by commas; only the first may have the dollar sign. The following forms are equivalent:

```
{$noindex,norange}
{$noindex}{$norange}
```

Embedded switches are counting switches. Each occurrence increments or decrements the switch value; the switch is enabled if its value is greater than zero. The initial value of a switch is controlled by an equivalent compilation switch, such as **debug**, if the equivalent compilation switch exists. If no equivalent switch is present on the command line, the initial value is determined by the defaults described below.

Once set, some switches are valid for the entire program, as with **$own**. In some cases, the "**no**" form of the switch is the one normally used, as with **$nomain**.

Some switches may be turned "on" and "off" for a particular section of code, either on a statement-by-statement or procedure-by-procedure basis. The following example shows how debugging may be turned off for a procedure:

```
  :
{$nodebug}—————————— debugging turned off

procedure P;
  begin
  :
  ———————————————————— body of procedure P
  :
  end;

{$debug}—————————— debugging enabled again
  :
  :
```

The compiler does not limit to the number of command-line switches you may use. When using embedded switches, you may have up to ten pairs of **$list...$nolist** switches and up to twenty-five "on...off" settings for other embedded switches.

The particulars of each switch are described in the following sections.

$double Specifies that all real arithmetic is to be done with double-precision rather than with single-precision. All **real** variables are in 8-byte floating-point format. **$double** applies to the entire compilation. You must use colon notation (e.g., E:18:15) to print the double-precision values in a **write** statement. This switch must appear in the program before any data of type **real** is defined or used. Default is "off"; with real variables in 4-byte format. See "Extended Precision" in the *Programmer's Reference* for more details.

$pascal1 Specifies that the interface with external procedures is compatible with Pascal–1. This interface is less efficient than that of Pascal–2, and may slow program execution, but should simplify conversion of programs from Pascal–1 to Pascal–2. The default is "off"; giving the Pascal–2 interface.

External Pascal–2 procedures may be called regardless of the setting of this switch.

$nomain No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is **$main**: a main program is being compiled.

$own Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with **$own**. The **$own** setting applies to the entire compilation unit. Default is "off": global variables are shared. With this switch, you can distinguish between "public global" and "private global" variables.

$nowalkback Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. The **debug** switch disables the generation of the walkback. Default is **walkback**: the tables are generated, and the full walkback in source terms is displayed after the run-time error message. See "Run-Time Error Reporting" later in this section for a discussion of the walkback. The **walkback** switch in external modules must be set to match those of the main program segment.

$nodebug, $debug Disables/enables some of the overhead of the Pascal-2 Debugger. These two switches have effect only when the **debug** compilation switch is specified. The **debug** switch generates the extra files needed for debugging and sets the **$debug** switch "on." **$Nodebug** may be used to turn off some of the debugging overhead for procedures or functions that have already been fully tested. **$Debug** may be used to restore debugging for other procedures. The **walkback** switch in external modules must be set to match those of the main program segment.

$noprofile, $profile Disables/enables some of the overhead of the Pascal-2 Profiler. These two switches have effect only when the **profile** compilation switch is specified. The **profile** switch generates the extra files needed for profiling and sets **$profile** "on." **$Noprofile** may be used to turn off profiling for procedures or functions that do not need to be profiled, and **$profile** may be used to restore profiling for other procedures.

When the **begin** statement of a procedure is compiled, the state of the **$debug/$nodebug** and **$profile/$noprofile** switches determine debugging or profiling for that entire procedure. Note that a procedure constitutes the smallest section of code that can be debugged or profiled; you can't debug or profile individual lines of a procedure.

The **$debug/$nodebug** and **$profile/$noprofile** switches serve the same functions as far as the code generated. You would never use both sets in the same compilation. (You can't debug the program and profile it at the same time.)

$nolist Turns off the listing of source lines in the listing file; **$list** restores the listing of source lines. The switch may be turned on or off after each line of source code. You may have up to ten **$list -- $nolist** switch pairs. The listing file displays the **$nolist/$list** switches, and the line numbers reflect the lines for which listing has been disabled. In this program fragment, listing has been disabled on lines 3 through 5:

```
1       program Ex(output);
2       {$nolist}
6       {$list}
7
8           begin
  :
```

Lines with errors are displayed even if the **$nolist** switch is on. Default is **$list**.

Do not use the **$nolist** switch during debugging sessions. If you attempt to access any "unlisted" line(s), the response is the message "No such statement in this procedure." Other errors also may be produced.

$standard Like the corresponding compilation switch, **$standard** causes all extended language features of Pascal-2 to be flagged as compilation errors. By using the embedded switch at the beginning of the program, you don't have to use the **standard** switch every time you compile the program.

In addition, if you want to compile the program using language extensions of Pascal-2, but you want to mark the non-standard features (for later transportability to another compiler, perhaps), insert the **$standard** switch at the start of the program, and enclose any non-standard sections with the switches **$nostandard** and **$standard**. The compiler then checks the rest of the program for non-standard features, so that you may minimize your use of extensions. The **$nostandard** switch is a textual flag to aid any future conversion to a standard program.

The **$standard** and **$nostandard** switches may be turned on or off after each line of source code. Default is **$nostandard**, which accepts the extended language features of Pascal-2 as correct forms.

Run-Time Checking Switches

The compilation switch **nocheck** turns off all run-time checks. There is no embedded **$nocheck** switch. The embedded checking switches cancel the particular checks listed below. Any of these switches may be placed at the start of the program to turn off a particular kind of check throughout. Or, "on/off" pairings may be used on a statement-by-statement basis within the program.

Turning off run-time checks reduces the size of the program. However, we recommend that you do not turn off any checks until the program has been fully debugged.

$nocheck Turns off all run-time checks.

$noindexcheck Stops generation of code for array bounds checks; no array index is checked as to whether it is within the array bounds. Default is **$indexcheck**.

$nopointercheck Stops generation of code that checks for **nil** or invalid pointer values. Default is **$pointercheck**.

$norangecheck Cancels the subrange assignment and **case** statement check capabilities. No assignment to a variable of subrange type is checked as to whether the assigned value is within the allowed range. Also, **case** selectors are not checked for matching labels. Default is **$rangecheck**.

$nostackcheck Stops the generation of code for stack overflow checks on procedure and function entry. No entry to a procedure or function is checked as to whether adequate stack space is available for local variables. Note that some procedures call support library routines that check for stack overflow. Thus, even when compiled with this switch, some programs may still report "stack overflow" errors. The default is **$stackcheck**.

Compilation Examples

The following examples show the effects of various switches on the compilation.

Example 1

```
>PAS PROG/LIST
```

Compiles the file PROG.PAS and generates an object file PROG.OBJ and a listing file PROG.LST. The check switch is assumed to be on, and code is generated for the hardware options of the machine on which the program is being compiled.

Example 2

```
>PAS PROG,PROG=PROG
```

Equivalent to Example 1.

Example 3

```
>RUN DB1:[100,10]PASCAL
PAS>PROG=PROG/NOCHECK/FIS
```

Compiles the file PROG.PAS and generates an object file PROG.OBJ. Any errors are listed on the user's terminal. No run-time checking code is generated, and code is generated for a CPU with FIS instructions.

Example 4

```
>PAS HEADER,MIDDLE,PROCED/NOMAIN
```

Concatenates and compiles the files HEADER.PAS, MIDDLE.PAS, and PROCED.PAS in the order given, and generates an object file, PROCED.OBJ. This code has no main body and therefore contains external procedures. The check switch is assumed to be on, and code is generated for the hardware options of the machine on which the program is being compiled.

Example 5

```
>PAS ,TI:=PROG
```

Produces a listing file to the terminal but no PROG.OBJ file.

Building an Executable Task

The Task Builder combines the main program with library routines from the Pascal and system libraries to produce an executable task (.TSK) image. Input to the Task Builder may also include external modules or libraries, overlay descriptions, and options that control memory and file allocation.

The basic Task Builder command (illustrated with a program called MAIN.PAS) is:

```
>TKB MAIN/FP/CP=MAIN,LB:[1,1]PASLIB/LB
```

This command combines the program MAIN.OBJ with the required modules from the Pascal library `LB:[1,1]PASLIB.OLB` and the system library `LB:[1,1]SYSLIB.OLB`, and produces the task image MAIN.TSK. The `/FP` switch directs the RSX system to save floating-point context information. ~~The~~ `/CP` switch designates the task as "checkpointable"; this means the task may be swapped to disk as necessary, and also that the task may be dynamically extended. The `/FP` and `/CP` switches are recommended for all Pascal tasks.

To include external modules, add the file names to the command line after the main program:

```
>TKB MAIN/FP/CP=MAIN,SUB1,SUB2,LB:[1,1]PASLIB/LB
```

Libraries of external modules may be included in a similiar fashion, but are marked with the `/LB` switch:

```
>TKB MAIN/FP/CP=MAIN,SUB1,LIB1/LB,LIB2/LB,LB:[1,1]PASLIB/LB
```

To produce a memory map which displays the contents of the task with the addresses and memory requirements of each component, add a second output file to the Task Builder command. The map file is created with the .MAP default extension.

```
>TKB MAIN/FP/CP,MAIN=MAIN,LB:[1,1]PASLIB/LB
```

Two Task Builder options commonly used with Pascal programs are `UNITS` and `EXTSCT`. These (and all) Task Builder options require the use of the multiline form of the `TKB` command, shown in the next example. The `UNITS` option increases the number of logical unit numbers (LUNs) available to the program. The number of LUNs available determines the maximum number of files which may be open at any time. Two LUNs (5 and 6) are always used by Pascal for the standard files `input` and `output`. There are 6 LUNs allocated by default, so a program using four or more files should allocate more LUNs with the `UNITS` option as follows.

(Additional UNITS cause a minimal increase in task size, so we recommend a large number, 20.)

```
>TKB
TKB>MAIN/FP/CP=MAIN,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=20
TKB>//
```

If used, the Debugger requires five LUNs for operation, so you must always increase the UNITS when you are using the Debugger with your program. See the Debugger Guide for details. LUNs are described in "Pascal-2's Use of LUNs" later in this guide.

The EXTSCT (Extend Section) option allocates additional memory for a program section. Pascal-2 uses the section named $$HEAP for the stack and local variables; if dynamic expansion is not available, the $$HEAP section is used for all buffers and the heap as well. The EXTSCT option parameters specify the section name and the number (in octal) of bytes of memory to allocate to that section. This example allocates 4K words to the stack:

```
>TKB
TKB>MAIN/FP/CP=MAIN,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>EXTSCT=$$HEAP:20000
TKB>//
```

The full capabilities of the Task Builder are described in the *RSX-11M/M-PLUS Task Builder Reference Manual*. See also the "Overlays" and "Monitoring Memory Usage" sections of the Programmer's Reference.

Using the Utilities

The programmer utility package contains a set of procedures and routines that enhance the capabilities of the Pascal-2 compiler. This subsection shows sample uses of three utilities: PASMAT formatter, the Debugger, and the Profiler. All utilities are fully explained in Sections 4 and 5 of this manual.

The Formatter

Suppose you have a program, EFACT.PAS, that calculates an approximation of e, the base of the natural logarithms, by summing the series

$$\sum_{i=0}^{n} \frac{1}{i!}$$

until additional terms do not affect the approximation.

Remember that the compiler accepts a program in whatever format you choose. So the program may look like this:

```
program Efact(output);
var E, Delta, Fact: real;
N: integer;
begin
E:=1.0; N:=1; Fact:=1.0; Delta:=1.0;
repeat
E:=E+Delta;
N:=N+1; Fact:=Fact*N; Delta:=1/Fact;
until E = (E+Delta);
write('With ', n:1, ' terms, ');
writeln('the value of e is',E:18:15);
end.
```

To make the program more readable, you decide to format the program with PASMAT, one of the Pascal-2 utility programs. Give the following command:

```
>RUN PASMAT
PMT>EFACT
```

and the program is reformatted to look like this:

```
program Efact(output);

  var
    E, Delta, Fact: real;
    N: integer;

  begin
    E := 1.0;
    N := 1;
    Fact := 1.0;
    Delta := 1.0;
    repeat
      E := E + Delta;
      N := N + 1;
      Fact := Fact * N;
      Delta := 1 / Fact;
    until E = (E + Delta);
    write('With ', n:1, ' terms, ');
    writeln('the value of e is',E:18:15);
  end.
```

(PASMAT has many other formatting options. See the Utilities Guide for details.) Now proceed to compile the program.

```
>PAS EFACT
>TKB EFACT/FP/CP = EFACT,LB:[1,1]PASLIB/LB
>RUN EFACT
With 11 terms, the value of e is 2.718282000000000
```

The Debugger

Even after you have corrected any syntax errors caught by the compiler, the program may still yield unexpected results. In this situation, Pascal-2's interactive Debugger can help uncover and correct the problems. The Debugger takes control of the program and responds to your commands, displaying execution information in a Pascal context. With the Debugger, you can watch the progress of the computation, and you can display intermediate values without making any program changes. You can then spot the point at which values go awry and correct the error.

To do this, use the **debug** switch to compile the program with the Debugger. You then build the task using the multiline form of the **TKB** command to increase the number of logical units available to the program. The support library and Debugger open seven files, so you need at least 7 logical units to run **Efact** and the Debugger. (The system default is 6.) Although 7 would suffice, we recommend that you use a larger number than you need, 20 in most cases.

First, compile and task-build the program with the commands:

```
>PAS EFACT/DEBUG
>TKB——————————— the multiline form of the command
TKB>EFACT/FP/CP=EFACT,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=20
TKB>//
```

The debug compilation produces four output files: EFACT.LST, EFACT.SYM, EFACT.SMP, and EFACT.OBJ. You need the listing file to determine the places to set breakpoints in the program. Don't worry about the other three output files, but don't delete them or the listing file. The Debugger uses all of them.

After doing a debug compilation, you may find it handy to have a printout of the listing file. The file looks like this:

Example: Listing From Compilation With Debug Option

```
Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM     Site #1-1     Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 EFACT/DEBUG

 Line Stmt
    1        program Efact(output);
    2
    3          var
    4            E, Delta, Fact: real;
    5            N: integer;
    6
    7     1    begin
    8     2      E := 1.0;
    9     3      N := 1;
   10     4      Fact := 1.0;
   11     5      Delta := 1.0;
   12     6      repeat
   13     7        E := E + Delta;
   14     8        N := N + 1;
   15     9        Fact := Fact * N;
   16    10        Delta := 1 / Fact;
   17            until E = (E + Delta);
   18    11      write('With ', n:1, ' terms, ');
   19    12      writeln('the value of e is',E:18:15);
   20          end.

*** No lines with errors detected ***
```

Two columns of numbers appear on the left side of each page. The first column, labeled `Line`, numbers each line of the source program. The second column, labeled `Stmt`, gives the statement number of the first statement on that line. The statement numbers start at 1 for each procedure or function, increasing by one as each statement is compiled. The Debugger uses these statement numbers to identify breakpoint locations in the compiled program.

In the program Efact, for instance, you may want to set a breakpoint at statement number 7. This is the point at which the approximation of e changes. If the program compiles correctly but produces unsatisfactory results, you may set the breakpoint at MAIN,7 to monitor the approximation to e as the program runs. We'll do just that in the next example.

Notice that the Debugger allows you to set the breakpoints. In this example, you tell the program to write the value of e at the breakpoint and then continue. (See the Debugger Guide for details on these commands.)

Example: Setting Breakpoints for Debugger

```
>RUN EFACT


Pascal Debugger V3.00 -- 12-Aug-83

Debugging program EFACT

} B(MAIN,7) <W(E);C> ———————— at breakpoint, write E and continue
} G———————————————————————— start program
Breakpoint at MAIN,7  E := E + Delta;
1.0000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.0000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.5000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.6666667E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7083335E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7166669E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7180557E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182541E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182789E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182817E+00
With 11 terms the value of e is 2.718282000000000

Program terminated.

Breakpoint at MAIN,12  writeln('the value of e is', E: 18: 15);
} Q——————————————————————————— quit
```

The Profiler

Finally, let's examine the program for efficiency by using the profile switch, which calls in the Profiler. "Profiling" shows the number of times each statement is executed, giving you the opportunity to optimize sections of code that are executed many times.

To utilize the Profiler, you must build the task using the multiline form of the TKB command to increase the number of logical units available to the program (same as for the Debugger). The support library and Profiler open seven files, so you need at least 7 logical units to run Efact and the Debugger. (The system default is 6.) Although 7 would suffice, we recommend that you use a larger number than you need, 20 in most cases.

Compile and task-build the program with the commands:

```
>PAS EFACT/PROFILE
>TKB———————— the multiline form of the command
TKB>EFACT/FP/CP=EFACT,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=20
TKB>//
```

Then execute the program. The Profiler takes control of your program and asks for the name of the profile output file. The default extension is .PRO.

Example: Screen Output From Program Executed Under Profiler's Control

```
>RUN EFACT



profile  V2.1  12-Aug-83

Profiling program: EFACT

Profile output file name? EFACT———————— Output goes to default extension
With 11 terms, the value of e is 2.718282000000000

Program terminated.

Profile being generated
```

The output is a listing file and looks like this:

Example: Profiler's Execution Summary

```
Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM     Site #1-1     Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202               EFACT/PROFILE

          Line Stmt
            1      program Efact(output);
            2        var
            3          E, Delta, Fact: real;
            4          N: integer;
            5
    1       6    1   begin
    1       7    2     E := 1.0;
    1       8    3     N := 1;
    1       9    4     Fact := 1.0;
    1      10    5     Delta := 1.0;
   10      11    6     repeat
   10      12    7       E := E + Delta;
   10      13    8       N := N + 1;
   10      14    9       Fact := Fact * N;
   10      15   10       Delta := 1 / Fact;
           16          until E = (E + Delta);
    1      17   11     write('With ', n: 1, ' terms ');
    1      18   12     writeln('the value of e is', e: 18: 15);
           19        end.

   *** No lines with errors detected ***

                        PROCEDURE EXECUTION SUMMARY

Procedure name    statements   times called   statements executed

MAIN                    12            1             57 100.00%

There are 12 statements in 1 procedures in this program.
       57 statements were executed during the profile.
```

The leftmost column of the profile listing shows the number of times each line is executed. The Profiler listing concludes with a "Procedure Execution Summary" that details each procedure name, the number of times it is called, the number of statements it contains, and the number of statements it executes. Note, too, that the summary shows the percent of execution count taken by each program block. (In this example, with only one procedure, the portion is 100%.) Given this information, you can attempt to optimize the procedures and statements that use a disproportionately large part of the time ("90 percent of the time on 10 percent of the program").

See the Profiler section of the Debugger Guide for more information and for a much more detailed example.

Detecting Run-Time Errors

The errors discussed so far have been compilation errors—errors detected by the compiler. Run-time errors, on the other hand, occur when a program is executing, after it has been compiler and linked.

Errors Detected at Run-Time

A run-time error such as `Array index out of bounds` stops the program at the point of error. The Pascal-2 error reporting system prints the error message, then traces the program's execution history, procedure by procedure, from the point of error back to the main program.

The error traceback, or "walkback," is intended to make debugging easier by showing precisely where the program stopped and which procedures were called to reach that point.

The following is an example of a run-time error and procedure walkback. (The program has already been compiled and linked.) Line numbers appearing in the walkback correspond to line numbers in the source listing, not line numbers in individual procedures.

```
$RUN CUSTOM ———————————— run program CUSTOM

%P2-F-SUBSCRIPT, Array index out of bounds —— error message

Error occurred at line 64 in procedure writelastname
 location of error
Last called from line 90 in procedure buildcustomerfile
Last called from line 103 in program customers
```

Your Next Step

Thus ends your guided tour through Pascal-2. At this point, you should be able to run a few simple programs. Before getting into complex programs, however, you should consult the Programmer Reference, the Language Specification, and study the guides for the Debugger and the Utilities, if you wish to use those options.

Pascal-2 V2.1/RSX Programmer's Reference

Introduction

The Programmer's Reference contains nitty-gritty information about Pascal-2 for programmers well-versed in the Pascal language. This reference describes, I/O control switches, and Pascal-2's low-level interaction with the PDP-11. This reference also describes ways to handle common Pascal-related implementation questions on RSX and contains other miscellaneous information.

This reference is not:

- an introduction to Pascal (see *Programming in Pascal* by Peter Grogono);
- a new user's guide to Pascal-2 (see the User Guide);
- a detailed description of Pascal-2 (see the Pascal-2 Language Specification).

I/O Control Switches

The `reset` and `rewrite` standard procedures accept two additional arguments specifying the file name of an external file and default fields of the file name. These arguments can also include I/O control switches that give explicit control of the operating system interface details. (A fourth parameter can also be specified. See the Language Specification for a complete discussion of `reset` and `rewrite`.)

The I/O switches appear in the file name or default name parameters as shown in these examples:

```
rewrite(F,'data/si:12','.dat/seek/span');
reset(F,Filename,'/rah/rw');
```

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters. The parameter *n* is a decimal number unless preceded by a crosshatch symbol (#), in which case it is an octal number.

`/aloc:n` `/cl:n` (Allocation or Clustersize): The parameter *n* is the number of 512-byte blocks allocated by the system each time a file is extended. The parameter *n* is also the cluster size. The default cluster size is set by the system manager and is usually 5 blocks of 512 bytes each. A positive value for *n* indicates a contiguous allocation; a negative value indicates a non-contiguous allocation. If *n* is not a multiple of the default, the system rounds the value up to the next highest multiple. The following statement creates a large file named TEST.TXT with a cluster size of 80 non-contiguous blocks. Later extensions of the file are allocated in 80-block clusters.

```
rewrite(F,'test.txt/cl:-80');
```

/blk (Blocked): Records in the file are not to cross disk block boundaries, allowing faster and easier access than unblocked records at the cost of additional space. This switch is the default for record files. See the **/span** and **/noblk** switches for the use of unblocked records.

/buff:*n* (Buffersize): The **/buff** switch specifies the storage to be allocated to a file buffer, with *n* representing the number of bytes. Pascal–2 normally allocates the minimum space required for a file buffer, 512 bytes. For disk files this default value may be raised by multiples of 512 to improve the efficiency of I/O transfers, at the cost of additional memory. Line-oriented files such as terminal and line-printer files normally receive buffer space equal to the width of the line, usually 80 characters. RSX single-character mode requires the buffer to be set to 1 byte, as shown in this example:

```
reset(input,'TI:/buff:1');
```

/cco (Cancel Control-O): Used with the terminal, this switch disables the Control-O (`^O`) mode and forces the output to the terminal. (Typing Control-O at the terminal disables output to the terminal until another Control-O is typed or until an input operation is performed.) A typical use would be to print an important error message even if Control-O had been typed.

/cr (Carriage Control): When printed, records in the file are to be preceded by a line-feed character and terminated by a carriage return. Valid only on **rewrite** statements opening disk files, this switch is useful for creating a printable file of raw data records. (For information on the opening of interactive files, see "Single-Character I/O" later in this guide.) For **text** variables this switch is automatically selected.

/cursor (Cursor Control): This switch enables terminal-independent cursor control, allowing Pascal programs to perform cursor positioning and graphics on a variety of terminals without the need for recoding for different terminals. The first two characters on each line are used for specifying the horizontal column and vertical line number at which the text is to be printed. Use the **chr** function to specify these special characters. Column 1 is the first print position of the line. Line 1 is the top of the screen. If 128 is added to the line number, the screen is cleared before the line is displayed. Based on these two characters, the terminal driver generates the necessary cursor control sequences to position the cursor.

/dlk (Don't Lock): This switch is provided for the rare occasions when the support library is unable to catch an I/O error and close the file (disk files only). Use this switch when you want to be sure that the file is closed after an error, allowing post-crash access to the file. Even with the **/dlk** switch, data in the file still may be destroyed.

/fta (FORTRAN Carriage Control): The first character of each record determines the line spacing before displaying the record, following the FORTRAN conventions. For example, a '1' in the first print position means a page eject before the rest of the record is printed. This switch may only be used with **text** files. For details on carriage control, see the section on "Terminal I/O" later in this guide.

/lun:n (Logical Unit Number): This switch allows you to specify the LUN for a specific file other than the default LUN assigned by Pascal–2. The switch is useful if you do not want to use Pascal to access files, for example if you are writing your own low-level I/O routines. This switch is also useful if a program calls FORTRAN routines that use files. In this way conflicts with LUN assignments can be prevented. For more information, refer to the section on Pascal–2's use of LUNs later in this guide.

/mbf:n (Multiple Buffering): Used with either the **/rah** or **/wbh** switch, this switch specifies the number of buffers (**n**) to use in accessing a file. The number of buffers to use with a file is dependent upon the file's I/O activity, the number of I/O operations performed by the program, the number of computations performed, and the amount of heap storage available to the program. Multiple buffering allows programs to overlap I/O operations with computations, significantly improving the performance of tasks that handle large volumes of data. (See "Multiple Buffering" later in this guide for information on enabling and utilizing the multiple buffering feature on your system.)

/nocr (No Carriage Control): This switch disables the automatic carriage controls imposed by a device (terminal, line printer, etc.). Line feed and carriage return characters are omitted from the file except in the case where the terminal is in **WRAP** mode (see the **/wal** switch, below). This switch is the default for non-**text** file variables.

/nsp (No Supersede): When a file is created with **rewrite** and **/nsp**, this switch causes an error if a file having the same name and version number already exists. If you specify a version number but omit **/nsp**, the contents of that version of the file is replaced with the new contents.

/rah (Read Ahead): Used with multiple buffers, this switch instructs the support library to read data from the file into the file buffer(s) before the information is actually needed by the program. Thus, program computations overlap file input operations, improving performance. (See "Multiple Buffering" later in this guide for information on enabling and utilizing the multiple buffering feature on your system.)

/ral (Read All Bits): This switch prevents characters from being interpreted on input from the terminal. In this binary input mode, carriage returns <CR> and escapes <ESC> do not terminate the line, and Control-C (^C) is not passed to the monitor. Instead, characters are read until the line buffer fills up. This switch is often used with the **/buff:1** I/O control switch to enable single-character input. For example:

```
reset(F,'ti:/ral/buff:1');
```

Any typed character is returned to the program, including Control-C.

/rcu (Restore Cursor): When this switch is used, the terminal driver saves the current coordinates of the cursor, prints the line of output, and then restores the cursor to its original position. This switch is commonly used in conjunction with the **/cursor** file switch to update a field on the terminal without disturbing input or output in progress elsewhere on the screen.

/rne
/noecho (No Echo): The terminal driver normally echoes each character typed as input unless the terminal has been set in no-echo mode with a **SET** command. The **/rne** or **/noecho** switch temporarily disables the echoing of input typed at a terminal. This no-echo mode is often used with single-character input mode (**/buff:1**). A good example use of these switches is to read a password from the terminal without printing (echoing) it as it is typed.

/ro (Read-Only): Only read accesses to the file are permitted. This switch is the default for files opened with **reset**. This switch permits several users to read the same file simultaneously, provided they have all used **/ro** to open the file. **/Ro** is sometimes accompanied by **/seek**.

/rst (Read With Special Terminators): This switch causes any non-printing character, such as control characters (e.g., **RUBOUT**, **ESC**), to terminate a line of terminal input. You can obtain the actual character that terminated the line from the file's I/O status block by using the file definitions in LIBDEF.PAS.

/rw (Read-Write): Both read and write access permissions are available. This is the default for files initially opened with **rewrite**. This switch is implied with **/seek**, and is required with **/apd** in order to append information to the end of the file (details below).

/seek (Direct-Access): The **/seek** switch permits use of the **seek** standard procedure, and allows both **get** and **put** operations on the file variable. **/Seek** implies the **/rw** switch. The **/seek/rw** switch combination allows you to update files opened with **reset**. For read-only files, you must specify **/ro** (read-only) to override the **/rw** default. The **seek** procedure is described in the Language Specification.

`/shr` (Share): This switch permits shared access to the file by multiple users. Note that Pascal–2 offers no built-in facilities for record locking/unlocking. You can use the predefined function `noioerror` to detect locked records. See "I/O Error Trapping" later in this guide for information on `noioerror`. To learn more about shared files, consult the section on "Shared Access to Files" in the *IAS/RSX-11 I/O Operations Reference Manual.*

`/si:n` (Size): This switch, used with `rewrite`, specifies the initial allocation of space for the file. The parameter *n* is given in blocks of 512 bytes. A positive value for *n* allocates contiguous blocks; a negative value allocates non-contiguous blocks.

`/spanned` (Spanned): Records in the file are allowed to cross disk
`/noblk` block boundaries, making most effective use of space. In files created or accessed by Pascal–2 programs, fixed-length records are normally "blocked." This means that an integral number of records are stored in one disk block of 512 bytes, with any remaining storage in that block being unused. The `/span` and `/noblk` switches cause records to be packed more efficiently, with records spanning from one disk block to the next. This action requires additional buffer memory, which is automatically allocated. Some extra computation also is needed.

Spanned and blocked files are not generally compatible. Files created with `/span` or `/noblk` should be read with the same switch.

`/temp` (Temporary): This switch marks the opened file for deletion upon `close` or at program termination. A file created with no file name, as in `rewrite(F)`, also is marked as a temporary file. Do not use `/temp` to delete existing files. Use the predefined procedure `delete` to do this. `Delete` is described in the Language Specification.

`/wal` (Write All Characters): This switch is used in sending binary output to a terminal-like device. With this switch, the terminal driver does not interpret the data. Line-wrap mode is disabled. This switch is useful with the `/nocr` file switch, where 8-bit data is written to a device connected to a terminal interface, such as a plotter, which is not really a terminal device. In this case, `/wal` signals binary output mode, and `/nocr` prevents the terminal driver from generating carriage-return and line-feed characters at the end of each line.

`/wbh` (Write Behind): With this switch, control returns to the program before a `write` or `put` operation has completed. This overlaps program computations with file output operations, improving program performance. (See "Multiple Buffering" later in this guide for information on enabling and utilizing the multiple buffering feature on your system.)

`/wbt` (Write With Break-Through): This switch causes the output to be written to the terminal regardless of its current state. The task must be privileged for this switch to take effect. The

`/PR:0` Task Builder switch should be used with the image file name in the `TKB` command line to create a privileged task. Be careful when using the `/wbt` switch because of possible side effects with screen-oriented editors.

The following switches permit access to more specialized capabilities of the File Control System. The descriptions refer to control fields in the File Descriptor Block (FDB), which is described in detail in Appendix A of the *IAS/RSX-11 I/O Operations Reference Manual.*

`/actl:`*n* Sets `F.ACTL` to the decimal value of parameter *n*; `F.ACTL` determines the number of retrieval pointers and magnetic tape positioning characteristics. For example, setting *n* to 34816 rewinds the tape before the next operation is performed on the tape, as shown:

```
reset(T,'mt0:rstape.dat/actl:34816');
```

`/apd` Sets `FA.APD` in `F.FACC`, indicating that records are to be appended to an existing file. This switch is used solely in conjunction with the `/rw` switch to open an existing file, as shown in this example:

```
reset(F,Filename,'/apd/rw');
```

`/ext` Sets `FA.EXT` in `F.FACC`, allowing extension of the file. Files opened with write access are by default opened with extend and read-write (`/rw`) access; you would rarely use this switch.

`/fix:`*n* Sets `R.FIX` in `F.RTYP`. This indicates a file of fixed length records of length *n*; this is the default record structure for non-`text` files. For `text` files opened with this switch, each record must be exactly *n* bytes long or an error results.

`/ins` Sets `FD.INS` in `F.RACC`, indicating that put operations in sequential mode should update the record and not truncate the file. This switch is automatically selected when `/seek` is used.

`/sq` Sets `R.SEQ` in `F.RTYP` to indicate a sequenced file; the sequence numbers are not readily available to the Pascal programmer.

`/var:`*n* Sets `R.VAR` in `F.RTYP` to indicate a file of variable length records, with a maximum length of *n* bytes. This switch can only be applied to `text` files. The default length for `text` files is `/var:132`. This switch is useful when you need to write lines of text longer than 132 characters; for example, `/var:256` indicates that lines in the file cannot exceed 256 bytes.

`/wrt` Sets `FA.WRT` in `F.FACC`, providing write but not extend access to an existing file.

External Modules

Pascal–2 implements separate compilation through the concept of an external module, a program fragment containing at least one procedure or function. External modules are compiled independently of other program units and combined by the Task Builder. External modules may be stored in libraries to simplify the handling of common routines. (See the section "Resident and Cluster Libraries.")

External modules may reference global variables shared by all of the modules making up a program. If each module (including the main program) is compiled with the same global variables, the effect is as if all modules were compiled together. For this to work properly, the global declarations in the external procedure and main program must contain the same variable declarations in the same order. Parameter lists also must agree (i.e., contain the same parameter declarations in the same order). The simplest and most efficient way to do this is to place all global declarations, including references to external procedures and functions, in a header file then include the header file in the external module and in the main program, using the `%include` compiler directive (see "Multiple Source Files").

External routines must be referenced at the outermost (global) level of a main program, but they may be called from any point in the code. An external module compilation requires either the `nomain` compilation switch or the `$nomain` embedded switch. Both switches specify that no main program is contained in the source file. The `nomain` switch is specified on the command line, whereas the `$nomain` embedded switch is placed at the beginning of the external module.

An external procedure name consists of the first eight characters of the procedure or function identifer. External procedure names must uniquely identify an external routine because they are used as global symbol names by the Task Builder.

Task-building errors most pertinent to external modules are:

- Duplication of external symbols. An external procedure has been defined in more than one module. When multiple definitions are encountered, the Task Builder uses the first definition only and ignores succeeding definitions.
- Undefined external symbols. The program has referenced an external routine that was not defined in any of the object files or libraries specified on the command line. This error indicates that the program contains unresolved global references.

In each case, the Task Builder responds with an error message and produces an output file that cannot execute (permission bits on the file are not set).

Two compiler directives, `external` and `nonpascal`, allow the use of external modules. The `external` directive defines a procedure or function implemented in Pascal–2 as "external," which means that the procedure may be referenced by other modules and that both the external module and the program or module that calls it expects to find the normal Pascal–2 calling sequence of parameters on the stack. The nonpascal directive defines a routine written in a language other than Pascal, such as FORTRAN or MACRO-11, and generates a call to the FORTRAN interface routine (P$111) in the Pascal support library. P$111 creates the standard DEC calling sequence of parameters which is expected by the external module, and which differs from Pascal–2's.

CAUTION

Observe two cautions when using the external or nonpascal directive. Parameters to external routines cannot be checked by the compiler for type conformance across module boundaries, so an accidental type mismatch may cause unpredictable results. Also, the compiler cannot verify the conformance of global data. As mentioned above, use of the %include directive can help reduce problems in this area. Parameters must be passed by reference.

Calls to Pascal–2 Routines

The syntax of the external directive is similar to the syntax of the forward directive in that it consists of two distinct parts: the declaration and the body. The declaration includes the external procedure or function name and the argument list, followed by the external directive.

```
procedure GetString(Arg:  Argtype);——— this is the declaration
    external;·——————— external directive is required
```

This declaration must appear in the external module and in each compilation unit that calls that external routine. You can place the declaration in a header file and use the %include directive to insert it appropriately.

The body of the external module contains the actual code for the procedure or function and must not include an argument list.

```
procedure GetString(Arg:Argtype);·—— this is the procedure body
  begin
:
:
:
:

  end;
```

The body and declaration must be compiled together in order for the external procedure to function properly. The external procedure may then be called in the same way as any other procedure or function:

```
GetString(Length);·——————— external procedure call
```

Example Using External Directive

The practical application of external procedures is best shown by example. The following sample illustrates the declaration and use of external procedures and the correct way to access global variables. Note that the global declarations must be identical in the external procedure (CHANGE.PAS) and in the main program (MAINLN.PAS). Note also the use of the `$nomain` embedded switch in the external procedure.

First, we create a separate header module HDR.PAS containing the external procedure reference and the program's global declarations.

Example: Separate Header Module (HDR.PAS)

```
type————————————————— global declarations
  GlobalType = record
      B: boolean;
      V: integer;
    end;
var
  Glob: GlobalType;
  I: integer;

procedure Change(P: integer);
  external;————————— external directive must appear
```

The file (CHANGE.PAS) consists of the external procedure `Change` and the `%include` directive, as follows:

Example: External Procedure (CHANGE.PAS)

```
{$nomain}—————————————————— embedded switch
%include hdr.pas—————————————— header module
procedure Change;————————————— no parameters here
  begin
    with Glob do
      begin——————————————— change global variables
        B := true;
        V := V + P;
      end;
  end;
```

The external procedure is then compiled with the `nomain` option embedded, using the command:

```
>PAS CHANGE
```

However, you can omit the embedded option and specify the `nomain` on the command line in the following manner:

```
>PAS CHANGE/NOMAIN
```

The externally defined procedure Change may now be called within any program unit that includes HDR.PAS and is task built with CHANGE.OBJ. For example, assume that the file MAINLN.PAS consists of this:

Example: External Procedures (MAINLN.PAS)

```
program Mainline;
  %include hdr;————————— external procedure and global declarations

  procedure Before;

    begin
      with Glob do
        writeln('Before executing Change, B = ',B,' and V =',V:2,'.');
    end;

  procedure After;

    begin
      with Glob do
        writeln('After executing Change, B =',B,' and V = ',V:2,'.');
    end;

 begin       { program Mainline }
   with Glob do
     begin————————————————————— initialize global variables
       B := false;
       V := 0;
     end;
   I := 45;
   Before;
   Change(I);———————————————————————— the external call
   After;
 end.
```

Compile MAINLN as you would any other main program and task build the main program and external module, as shown:

```
>PAS MAINLN
>TKB MAINLN/CP/FP=MAINLN,CHANGE,LB:[1,1]PASLIB/LB
```

Running the program yields these results:

```
>RUN MAINLN


Before executing Change, B = false and V = 0.
After executing Change, B = true and V = 45.
```

Calls to Non-Pascal-2 Routines

The **nonpascal** directive is used instead of **external** when the external procedure is generated by an assembler or a compiler other than Pascal-2. **Nonpascal** creates an interface between the Pascal-2 calling sequence generated by the main program or module and the standard DEC calling sequence required by FORTRAN and most MACRO-11 routines. In addition, when the **nonpascal** directive is invoked, register **R5** is used as a pointer to the list of parameters. Parameters must be passed by reference.

Syntax for the **nonpascal** directive consists of a separate declaration and body. The declaration contains the name of the procedure or function and the argument list, followed by the **nonpascal** directive.

Calling MACRO Subroutines

The external declaration for a MACRO function looks like this:

```
program Test;

var i: integer;

function afunct (var i: integer) : integer; — declaration of
  nonpascal; ——————————————— MACRO routine

begin ——————————————— body of the main program
  i := 10;
  writeln(afunct(i));
end.
```

The MACRO routine AFUNCT is written:

```
; Returns argument plus 10


AFUNCT::
      mov    @2(r5),r0
      add    #12,r0
      rts    pc
      .end
```

Type matching for the declaration and use of parameters are the user's responsibility.

The sample program **Test** is compiled with the command:

```
>MACRO AFUNCT
>PAS TEST
>TKB TEST/CP/FP=TEST,AFUNCT,LB:[1,1]PASLIB/LB
>RUN TEST
      20
```

MACRO routines written with the Pascal-2 PASMAC utility must be declared as **external** rather than **nonpascal**, because PASMAC simulates the Pascal-2 calling sequence.

Calling FORTRAN Subroutines

For Pascal programs to call FORTRAN subroutines, two conditions must be met: all parameters must be passed by reference, not by value, and FORTRAN subroutines must be declared in Pascal programs with the **nonpascal** directive.

The procedure **forini**, supplied in the Pascal support library, initializes the FORTRAN object library, allowing Pascal programs to call FORTRAN subroutines. The FORTRAN object library must be initialized in case the called FORTRAN subroutine calls other subroutines in that library. If the library is not properly initialized, traps or inaccurate executions can result. The same initialization is required for FORTRAN subroutines called from MACRO. After **forini** returns, you should be able to call almost any FORTRAN subroutine.

Declare **forini** as shown:

```
procedure Forini;       { FORTRAN initialization routine }
  external;
```

Call the FORTRAN initialization procedure at the first convenient point in your code, preferably the first statement of the main program.

The program FTEST.PAS shows a way to call FORTRAN subroutines from a Pascal program. The program reads three integers from the terminal, calls the FORTRAN subroutine ADDEM to calculate the sum of the three numbers, then prints the sum.

Subroutine ADDEM.FOR contains these statements:

```
      SUBROUTINE ADDEM(A,B,C,D)
C
C  ADDS A, B, C TO PRODUCE D.
C
        IMPLICIT INTEGER (A-Z)

        D = A + B + C

        RETURN

      END
```

The main Pascal program contains these lines:

Example: Program With Call to Fortran Subroutine

```
program FORtest;
  var
    A, B, C, D: integer;

    procedure ADDEM(var A, B, C, D: integer);
      nonpascal;

  begin
    write('Enter 3 values: ');
    readln(A, B, C);
    ADDEM(A, B, C, D);    { add the numbers}—— FORTRAN call
    writeln('The answer is ', D);
  end.
```

Compile and run the program using these commands. Assume that the FORTRAN subroutine ADDEM has already been compiled.

```
>PAS FTEST
>TKB FTEST/CP/FP=FTEST,ADDEM,LB:[1,1]PASLIB/LB
>RUN FTEST
Enter 3 values: 20 40 120
The answer is      180
```

Two restrictions relate specifically to the use of FORTRAN subroutines in Pascal programs. First, Pascal-called FORTRAN subroutines cannot access files opened in the Pascal program. However, these FORTRAN routines can use files that they themselves open.

Second, FORTRAN allows the passing of "null" parameters to subroutines, in which a comma is used as a place holder for an optional parameter. Pascal has no such feature. To pass null parameters to a FORTRAN subroutine from Pascal, use the **origin** directive to declare the null parameter. The variable and procedure declaration for the Pascal program is shown below. You may use the same **origin** declaration of a null real parameter for **real** and **integer**.

```
var
  ListNumber, LastList: integer;
  Null origin 177777B: integer;———— null integer parameter
  Rewind: boolean;

procedure FPREW(var Number, Last: integer;
                var I: integer; var R: real;
                var Rew: boolean);
nonpascal;
```

The FORTRAN subroutine declaration is then:

```
subroutine FPREW(Number,Last,I,R,Rew)
```

When you call the FORTRAN subroutine from the Pascal program, substitute the appropriate **Null** variable for any unnecesary parameters. In the case of FPREW, the third and fourth parameters are null parameters:

```
FPREW(ListNumber,LastList,INull,RNull,Rewind);
```

External Module Libraries

Suppose you want a library of procedures that can be referenced by any program. For a particular program, you do not necessarily reference all the procedures in that library, and you do not want the entire library loaded with the program.

Procedures and functions from one compilation unit form a single object module and cannot be selectively loaded. For example, if procedures `A`, `B`, and `C` are compiled together and placed in a library, any reference to one of them causes all three to be loaded. On the other hand, if each procedure `A`, `B`, and `C` is compiled separately and the three object modules are placed in the same library, then a reference to one of them causes only that module to be loaded in the program. To keep final program size to the minimum, library modules should be compiled separately whenever possible.

Rather than having an external declaration in the main program for each procedure needed, create a single "header" file containing the external declarations for all the external procedures defined in the library. This header file can be included in the compilation with the `%include` directive placed near the beginning of the program source file. No external reference is generated for any external procedure in the header file that is not used by the program, so only those modules actually used by each compilation unit are loaded into the final image file (assuming that the library modules modules were themselves separately compiled). See "Multiple Source Files" for use of the `%include` directive.

By using a header file in this way, you can avoid errors that could be caused by a mismatched declaration, forcing any change made to a declaration for an external procedure to be reflected in all programs using that procedure. Carried to the fullest extent, a library and its corresponding header file can be used system-wide.

Extended Precision

Values of type `real` are normally stored in the PDP-11 single-precision format, which requires 2 words of storage per value and offers about 7 decimal digits of precision. The `double` compilation switch or the `$double` embedded switch gives double precision to all `real` values. Each extended-precision value occupies 4 words of storage and provides approximately 15-digit precision in all `real` calculations, including the transcendental functions.

Normal and extended-precision values cannot be mixed in a program: the `double` or `$double` switch generates extended precision for all `real` values. All external modules must be compiled with the same precision as the main program, even if no `real` variables are present.

In addition, you must use the colon notation output format (e.g., `E:18:15`) to display double precision values in `write` statements.

Overlays

The Task Builder is capable of creating overlaid tasks, wherein program memory areas not in use can be shared by other sections. The use of overlays helps to reduce the memory requirements of a program, because the entire program does not have to reside in memory.

An overlaid program is divided logically into "segments." When one segment is executing, the unused segments are stored in the task image on the disk. When the executing segment finishes its work, it is overwritten with the next segment to be executed.

Overlays may need to be used if run-time errors such as "not enough memory" or "stack overflow" cause the program to abort. The examples presented here give only a brief overview of the RSX overlay capabilities. If you have any problems, see chapters 5 and 6 of the *RSX-11M/M-PLUS Task Builder Reference Manual* for further details and examples; this overview is oriented only toward Pascal tasks.

The overlay structure of a program can be complicated, so a special "language" exists to define overlays. That language is the Overlay Description Language (ODL). An overlay structure is defined in an ODL file describing each program segment and its position in an overlay tree. The ODL file has the same name as the root segment, with the extension .ODL (i.e., MAIN.ODL is the ODL file for MAIN.PAS).

The overlay description file `LB:[1,1]PAS.ODL`, supplied with Pascal-2, contains overlay descriptions for the Pascal run-time library and the File Control Services (FCS) I/O routines. This file describes the way you overlay the modules in the Pascal support library and the system I/O library. Overlaying these two libraries is your first step toward conserving space. The examples that follow show you how to use PAS.ODL to do this. If, after overlaying these libraries, you find that you still need to shave the size of a program, then try to split up your code into overlays.

These overlay structures are defined in PAS.ODL:

`ROOT`	The essential support library routines that reside in the root co-tree. This co-tree must always be specified. (Co-trees are discussed in the Task Builder reference manual.)
`LIBR`	The remainder of the support library routines. This co-tree must always be specified.
`SYSIO`	A co-tree for the FCS I/O routines used by Pascal. This co-tree must always be selected.
`SINGLE`	The library routines for single-precision `real` arithmetic and transcendental operations. Select this co-tree if your program uses single-precision `real` variables or procedures.
`DOUBLE`	The extended-precision library routines. Select this co-tree if your program uses double-precision `real` variables or procedures, and is compiled with the `double` compilation switch.

DEBUG2 The Pascal–2 Debugger. Select this co-tree if your Pascal–2 program was compiled with the debug compilation switch. When you specify this co-tree in an ODL file, the DOUBLE co-tree is automatically selected to satisy the Debugger's need for double-precision arithmetic. For programs using single-precision, you have to specify SING, a special variant of the SINGLE co-tree, designed for use with DEBUG2. SING accounts for the fact that the Debugger is itself a Pascal program compiled with double precision.

To illustrate the use of these structures, consider a Pascal program named DEMO.PAS. If the program uses integer arithmetic only (no real numbers), the program's ODL file contains:

```
@LB:[1,1]PAS.ODL
        .ROOT    ROOT-DEMO,SYSIO,LIBR
        .END
```

The first line references PAS.ODL as an indirect file. This file is always referenced in overlay descriptions. The .ROOT directive in the second line defines the co-trees selected for the program. ROOT, SYSIO, and LIBR must always be selected. The hyphen '-' is used to place your program (DEMO) in the root. The main program must be in the root because DEMO.OBJ contains the transfer address for the task. The .END directive signals the end of the overlay description.

The following command builds the overlaid task DEMO.TSK. On the command line, the ODL file is specified with the /MP switch. This command uses the overlay description file DEMO.ODL to produce the task DEMO.TSK and the map file DEMO.MAP (the second output file). DEMO.ODL references the object file DEMO.OBJ as well as the Pascal overlay file PAS.ODL.

```
>TKB DEMO/FP/CP,DEMO=DEMO/MP
```

When using the Debugger, you must increase the number of logical unit numbers (LUNs) available to your program when you task-build the program, as shown below. The default number of LUNs is 6. Since the Debugger opens several files of its own, we recommend that you increase the number to 20, to be sure your program can open all its files. (Extra LUNs use very little memory.)

```
>TKB
TKB>DEMO/FP/CP,DEMO=DEMO/MP
ENTER OPTIONS:
TKB>UNITS=20
TKB>//
```

Examples of ODL Files

In the following examples, substitute the name of your program for the program name DEMO.

Example 1. Overlaying a program that uses single-precision arithmetic.

```
@LB:[1,1]PAS.ODL
         .ROOT    ROOT-DEMO,SYSIO,LIBR,SINGLE
         .END
```

Example 2. Overlaying a program that uses double-precision arithmetic.

```
@LB:[1,1]PAS.ODL
         .ROOT    ROOT-DEMO,SYSIO,LIBR,DOUBLE
         .END
```

Example 3. Overlaying a program that uses single-precision and /debug.

```
@LB:[1,1]PAS.ODL
         .ROOT    ROOT-DEMO,SYSIO,LIBR,SING,DEBUG2
         .END
```

Example 4. Overlaying a program that uses double-precision and /debug.

In this situation you do not need to specify the DOUBLE co-tree, because it is automatically selected by the DEBUG2 co-tree.

```
@LB:[1,1]PAS.ODL
         .ROOT    ROOT-DEMO,SYSIO,LIBR,DEBUG2
         .END
```

Example 5. Overlaying external modules.

Suppose that DEMO.PAS calls three separate modules, SUB1.PAS, SUB2.PAS and SUB3.PAS. If the procedures in SUB1, SUB2 and SUB3 do not call each other you may overlay them, as shown:

```
@LB:[1,1]PAS.ODL
         .ROOT    ROOT-DEMO-*(SUB1,SUB2,SUB3),SYSIO,LIBR
         .END
```

The parentheses indicate that the modules SUB1.OBJ, SUB2.OBJ, and SUB3.OBJ are to be overlaid against each other (they share the same memory). The asterisk (*) indicates that the automatic loading mechanism should be used.

Example 6. Using complex overlay descriptions.

If the overlay description is complex and does not fit on one line, you can break up the definitions using "factors." In this example, the factor USER is defined by the .FCTR directive. The USER factor is then used in the .ROOT directive. This example is equivalent to the previous example.

```
@LB:[1,1]PAS.ODL
USER:    .FCTR    DEMO-*(SUB1,SUB2,SUB3)
         .ROOT    ROOT-USER,SYSIO,LIBR
         .END
```

Support Library

The Pascal support library is a collection of modules contained in an object module library called PASLIB.OLB located in LB:[1,1]. When compiling a program, the Pascal compiler generates subroutine calls to routines in the Pascal support library. The Task Builder places these routines in the psect P$LIBR. The entry points in the library are identified as p$*nnn* where *nnn* is a small integer. Appendix E of this guide contains a list of these support library entry points. To see these subroutine calls, inspect the MACRO-11 code generated by the Pascal-2 macro switch. Support library routines not called by the compiler have a name instead of a number following the p$.

Most of the routines in the Pascal support library perform I/O operations or arithmetic computations such as floating-point simulation or trigonometric function approximation. Other routines allocate dynamic memory and report error conditions. Still other routines allow you to change the run-time error reporting to suit your needs. When you build a Pascal task, the Task Builder searches the Pascal support library for the modules required to run your task. For example, if you compute a logarithm in your program, the Task Builder includes the support library module that approximates logarithms ($FLOG, which defines the entry point p$102).

In most Pascal tasks, the Task Builder includes from 3K words to 9K words of library modules.

Initializing the Support Library

The Pascal support library is initialized at the start of a Pascal program. When a typical execution begins, the system transfers control to p$bgn, the transfer address of the program, and the support library initialization procedure p$59 is called. This procedure initializes global variables used by the support library in module $DATA; then it uses the EXTK$ system directive to expand the program code by 4K bytes to make room for the stack, which is originally located in low memory. The routine then moves the stack from low memory to its new location at the end of the program code.

After the stack is repositioned, control transfers to p$33, the file initialization routine. P$33 assigns the standard files input and output (logical unit numbers 5 and 6) to TI:, the terminal. (Logical unit numbers are discussed a later subsection and in "Pascal-2's Use of LUNs" later in this section.) The support library then transfers control to the first statement of the program, and execution begins. However, if the program is being debugged with the Debugger, control transfers to p$67, the Debugger initialization routine, instead of the program. This routine initializes the Debugger and opens its files. The Debugger then takes over execution of the program. (See the Debugger Guide for details.)

Support Library Data Definitions

Constants, data and internal file definitions used by the support library are contained in the file LIBDEF.PAS, included in the Pascal–2 distribution kit. In addition to its use with the support library, this file is "included" in the installation verification program VERIFY.PAS and the error-reporting module OPERRO.PAS. The LIBDEF routine defines the file name block, the file descriptor block and related characteristics, the file variable, and the library work area (psect **$$VEX1**).

By using the **%include** directive, users can include LIBDEF.PAS in any program that accesses the support library's work area directly. The example program PFDB.PAS, below, defines a function named **GetFDB**, which returns the address of the RSX File Descriptor Block (FDB) as an integer. The FDB is used by the File Control Services (FCS) to perform I/O operations on the file. The address of the FDB for a Pascal file can be used to perform special I/O functions such as spooling the file to the line printer. (See the *RSX-11M/11M-PLUS I/O Operations Reference Manual* for more information.) Also note the use of the function **loophole**, which is predefined in the compiler and is used in **GetChannel** (and **GetLUN**). See the Language Specification for details on **loophole**.

Example: Program to Print FDB Address (PRINTFDB.PAS)

```
program PrintFDB;
%include p:libdef;

{ Returns the address of the FDB for a Pascal file.  Note: Record devices
  such as terminals and line printers do not have FDB's. }

  var
    X: text;
    Filename: packed array [1..10] of char;

    function GetFDB(var X: text): integer;
      var
        F: user_file_variable;------------ data type defined in LIBDEF.PAS

      begin   { procedure GetFDB }
        F := loophole(user_file_variable,X);
        GetFDB := loophole(integer, F^.fdb);
      end;   { procedure GetFDB }

  begin  { program PrintFDB }
    write('Enter a file name: ');
    readln(Filename);
    reset(X,Filename);
    writeln('The address of ',Filename,'''s FDB is: ',GetFDB(X));
  end.   { program PrintFDB }
```

To prepare the program for execution, use these commands:

```
>PAS PFDB
>TKB PFDB/FP/CP=PFDB,LB:[1,1]PASLIB/LB
>RUN PFDB
Enter a file name: TEST.TXT
The address of TEST.TXT  's FDB is:    26258
```

LIBDEF.PAS can also be used to determine the logical unit number (LUN) of a file already open to a program. Function GetLUN, supplied below, can replace procedure GetFDB in the above example so the program can print the LUN associated with the file.

```
function GetLUN(var X: text): integer;
  var
    F: user_file_variable;-—— data type defined in LIBDEF.PAS

  begin   { procedure GetLUN }
    F := loophole(user_file_variable,X);
    GetLUN := loophole(integer, F^.lun);
  end;    { procedure GetLUN }
```

NOTE

The definitions in LIBDEF.PAS are provided for informational purposes and are subject to change with each release of Pascal-2. Users who desire a more detailed description of the internal workings of the Pascal support library must obtain the library sources.

Modifying the Support Library's Global Variables

Certain global variables initialized in the Pascal support library can be changed at task-build time to fit the needs of a particular application. The GBLPAT (Global Patch) Task Builder option provides this flexibility. Pascal-related patches you can perform with the GLBPAT option are:

- You can assign the standard files input and output to different logical unit numbers (LUNs) than the default.
- You can change the event flag number used by the support library.
- You can prevent your program from being automatically extended.
- You can prevent your program from attaching to the terminal (TI:).

To use the GLBPAT option, you have to know the name of the root segment of the program. Determine this from the Task Builder map. The examples that follow assume that the root segment is PROG. See the *RSX-11M/11M-PLUS Task Builder Manual* for more details of the GLBPAT option.

Assigning Input and Output to Different LUNs

The standard files input and output are opened on LUNs 5 and 6, respectively, during program initialization. These files can be reassigned to other LUNs at task-build time with the GLBPAT option. This reassignment is necessary if you have an existing program (written in Pascal, FORTRAN or MACRO-11) that already uses these logical unit numbers for other files. The global integer variables p$ilun and p$olun, defined in the module $DATA in the Pascal support library, determine the LUNs used for input and output, respectively. By default, p$ilun is '5' and p$olun is '6.'

For example, if you want input to be read from LUN 1 (instead of 5) and output to be written to LUN 4 (instead of 6), use the following commands:

```
>TKB
TKB>PROG/FP/CP=PROG,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>GBLPAT=PROG:P$ILUN:1
TKB>GBLPAT=PROG:P$OLUN:4
TKB>//
```

See "Pascal-2's Use of LUNs" later in this guide.

Changing the Support Library's Event Flag

When the support library access a file, two different event flags are used depending on the type of device being accessed. For I/O to a directory-structured device such as a disk, the support library uses the File Control Services (FCS) to access the file. FCS uses event flag number 32 to synchronize the disk read and write operations. Users must avoid referencing EFN 32 in their programs. (Refer to the *RSX-11M/M-PLUS I/O Operations Reference Manual* for more information on the way FCS uses event flags.) For I/O to a record-oriented device such as a terminal or line printer, the support library accesses the device directly using the QIO (Queue I/O) system directive.

The support library accesses event flag number 17 to test completion of an I/O operation to a record-oriented device. In case you have an existing program or package of programs that uses EFN 17, you can reassign the EFN used by the support library to any unused event flag number, rather than changing the program code to use another EFN. The GBLPAT Task Builder option permits this. The event flag number is stored in the global variable named p$efn in the module $DATA in the Pascal support library.

The following TKB command illustrates the setting of p$efn to 5. This command instructs the support library to use event flag number 5 instead of 17 when performing I/O operations on record-oriented devices.

```
>TKB
TKB>PROG/FP/CP=PROG,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>GBLPAT=PROG:P$EFN:5
TKB>//
```

See the section on "Using Event Flags" later in this guide for information on using event flags from Pascal programs. Also see *RSX-11M/M-PLUS Executive Reference Manual* for complete details on the use of event flags.

The "No-Extend" Patch

The global symbol p$next can be patched to be a NOP (No Operation, octal 240) instruction, thus preventing Pascal programs from automatically asking RSX for more memory when no space is available on the free list. This modification may be needed in these circumstances:

- When resident overlays are used (see the *RSX-11M/M-PLUS Executive Reference Manual* and the Task Builder manual);
- When a Pascal task is mapped to shared common libraries;
- When PLAS directives are being used to change the mapping of virtual memory.

This patch prevents your program from extending into virtual address spaces being used for other things. If you use this patch, you must also explicitly allocate space for $$HEAP with the EXTSCT Task Builder option, as shown:

```
>TKB
TKB>PROG/FP/CP=PROG,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>GBLPAT=PROG:P$NEXT:240
TKB>EXTSCT=$$HEAP:20000
TKB>//
```

The "No-Attach" Patch

All Pascal tasks attach to the user's terminal (logical name TI:) when initialized. You can prevent the Pascal support library from attaching to TI: by patching the global symbol p$natt to be a NOP instruction (octal 240). This patch is useful if you wish to execute additional tasks from the terminal while the first task is executing. This patch also allows your program to execute MCR command lines from within the Pascal program. (See "Executing MCR Commands from Pascal Programs" later in this guide.)

For the root segment PROG, the patch is:

```
>TKB
TKB>PROG/FP/CP=PROG,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>GBLPAT=PROG:P$NATT:240
TKB>//
```

For related information, see "Detaching From the Terminal" later in this guide.

Setting the Length of the Compiler's Listing File

With the GBLPAT Task Builder option, you can change the number of lines the compiler prints on each page of the listing file it generates. Make the change permanently by modifying the command file PASCAL.CMD, which builds the Pascal compiler. The GBLPAT option, in this case, patches the location PAGEL$+2 to contain the desired number of lines, not including header lines. The format of the patch is:

```
>TKB
TKB>PROG/FP/CP=PROG,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>GBLPAT=HEAP:PAGEL$+2:n
TKB>//
```

where *n* is the page length in octal lines of source code. The value '66' is the default page length (54 decimal). If your line printer uses a different page size, adjust the value 66 up or down as necessary.

Run-Time Organization

The run-time organization of a Pascal-2 program is determined by the compiler, linker, and Pascal-2 support library.

You control compilation with the **pas2** command. By default, **pas2** instructs the compiler to produce an object file for your program. The linker combines the object output from the compiler with other object modules you specify on the command line and with modules called by the program from the Pascal-2 support library. The linker stores the executable output. When you run your program, the operating system then loads the file into physical memory and begins running the program.

Form of the Generated Code

Pascal-2 code is divided into program sections called "psects." The psects for the main program and any separately compiled procedures are combined with the Pascal support library by the Task Builder to produce an executable task image. The use of multiple psects arranged in alphabetical order provides greater flexibility for the combination of individual procedures into a program.

The compiler generates these psects:

CONSTS Contains all constants generated by the compiler. This includes constants declared by constant declarations or implicit in the code. This psect also contains jump tables generated by case statements, so there is a complete separation of instruction references and data references. The **CONSTS** psects for all compilation units are concatenated; compiled code does not attempt to write to this psect.

DIAGS Contains line number and procedure name data used in the printing of the run-time walkback. The information is encoded to save space. This psect is not generated if the **nowalkback** switch is specified on the command line.

GLOBAL Contains all global variables used in the main program and external procedures. This psect is arranged so that the global variables are shared among all procedures. The main program and all procedures that reference global variables should have exactly the same declarations. The size of the resulting psect is that of the largest **GLOBAL** psect generated by any of the compilation units.

If the **own** switch is specified in the compilation, this psect is instead named with the first six characters of the program name, allowing multiple global variable segments. Compiled code writes to this psect.

P$CODE Contains the instruction code for the compilation unit. The **P$CODE** psects for all compilation units are concatenated; compiled code does not attempt to write to this psect.

P$DYNL Defines a dynamic link to the Post-Mortem Analyzer, which prints the walkback. This psect has two words. The first word is a pointer used by the PMA to trace the stack frames for the walkback. With walkback enabled, the second word of this psect contains the address of P$PMA, the entry point of the PMA. If the nowalkback or nomain compilation switch is used, the second word contains a zero and no jump is made to the PMA.

SHIFTS	Generated only if the target machine does not have the EIS hardware option (sim). This psect contains a table of shift instructions that simulate multiple shifts. The psect is overlaid in a manner similar to TABLES and is treated as read-only by the compiled code.
TABLES	Contains bit tables used for access to set elements and individual bits within a word. All Pascal compilations generate this psect, but all copies are overlaid by the Task Builder so that only a single copy exists in the final program. Compiled code does not attempt to write to this psect.

The following table summarizes the attributes of the various psects. Refer to the MACRO-11 manual or the RSX Task Builder Manual for further information on the meaning of the attributes.

Psect name	Attributes
CONSTS	RO,D,LCL,REL,CON
DIAGS	RO,D,LCL,REL,CON
GLOBAL	RW,D,GBL,REL,OVR
P$CODE	RO,I,LCL,REL,CON
P$DYNL	RW,D,GBL,REL,OVR
SHIFTS	RO,I,GBL,REL,OVR
TABLES	RO,D,GBL,REL,OVR

So that Pascal programs may be included in libraries, each Pascal-2 object file has a module name consisting of the first six characters of the output file name. Thus a program compiled with the line:

```
>PAS RESPROG = HDR,INPROG
```

has the module name RESPRO.OBJ. This compilation performs "source concatenation." Note that with source concatenation INPROG.PAS must not contain a program statement or compilation errors result.

Memory Organization

On the PDP-11, a program has access to 32768 words (frequently abbreviated to 32K). The exact arrangement of storage is determined by the commands to the Task Builder, but a typical program may look something like Figure 2-1, which represents a snapshot taken during execution. The numbers are representative; actual values vary from program to program.

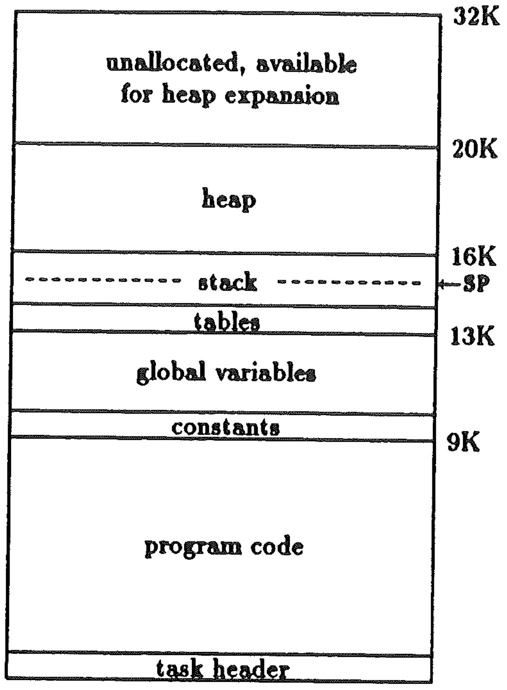


Figure 2-1. Typical Memory Layout of a Pascal Program.

Task Header The task header contains task parameters and data required by the executive and provides a storage area in which the task context is saved.

Program Code The program code section contains the instructions for the user program, plus any support library modules that may be required. The support library may add from 3K to 9K words of overhead. The size of this section is determined by the amount of user code.

Constants The constants section consists of all constants, such as strings or real constants, needed by the program. The section also contains the jump tables for case statements. The size of this section is determined by the user code.

Global Variables The global variables section contains the global variables used by the Pascal main program and external procedures. The size is that of the largest global variable section in any compilation unit.

Tables The tables section, which contains data needed by all Pascal programs, is 40 bytes long.

The Stack

The stack contains all variables local to inner blocks of the program, plus parameters, procedure linkage information, and temporary working storage. Upon entry to a procedure or function, space is allocated on the stack (a stack frame) containing space for all storage local to that block. The format of the stack frame is described below.

The space allocated for the stack is determined by the size of the psect $$HEAP. If this psect has a size of zero (the default value) then the task is extended at program initialization by 2048 words and the space so obtained is used for the stack. If $$HEAP has a non-zero size, then the space provided is used for the stack. The psect size may be set at task-build time with the option:

```
EXTSCT=$$HEAP: n
```

where *n* is the amount of space (in octal bytes) to allocate to the psect $$HEAP. A program that makes heavy use of the stack (for local variables or recursion) requires explicit allocation for $$HEAP.

If the task cannot be extended for some reason, space for the stack and heap must be provided via the EXTSCT command. For example, the heap must be extended in this way if the Pascal task is not checkpointable.

The stack pointer (SP) always points to the top of the stack (lowest physical space). If the space available for the stack is too small, the stack pointer eventually exceeds the limits of the stack space and causes the "stack overflow" error.

If you see this error, try extending the heap as described above.

WARNING

Especially large stack overflows, those that extend beyond the heap and into the program code sections of memory, have the potential for causing serious problems. In some cases it is possible for the error to prevent the appropriate error message from being printed or the program from properly terminating. In some rare instances, the condition can even lead to the disruption of the computer's operating system. It is the programmer's responsibility to avoid such excessively large overflows by controlling the size of local variables and value parameters passed to procedures.

The Heap

The heap is an area for dynamically allocated memory used for I/O control blocks, buffers, and variables allocated with `new`.

If the task can be expanded, space for the heap is expanded as necessary. If no space is available for further expansion or if the task cannot be expanded, space is taken from the bottom of the stack area. Thus the `$$HEAP` psect can be used for both the stack and the heap.

Space is returned to the heap when files are closed or when variables previously allocated with the standard procedure `new` are deallocated with the standard procedure `dispose`. Such space is then available for further heap allocation. The error message "not enough memory" results if no space is available to satisfy a request for heap storage.

Dynamic task expansion as described above requires that the Extend Task (`EXTK$`) directive be included during RSX system generation and that the task be checkpointable (`/CP` task-build switch). If the task cannot be dynamically expanded, the `EXTSCT=$$HEAP:` *n* option must be specified at task-build time to allocate space for the stack and heap.

For information on ways to monitor the allocation of the stack and heap at run-time, refer to the section on "Monitoring Memory Usage" later in this guide.

The Stack Frame

As each procedure or function is entered, space is allocated on the stack for parameters, linkage data, and local use. This space is called a "stack frame"; the "stack" consists of these stack frames.

Figure 2-2 shows the format of a stack frame.

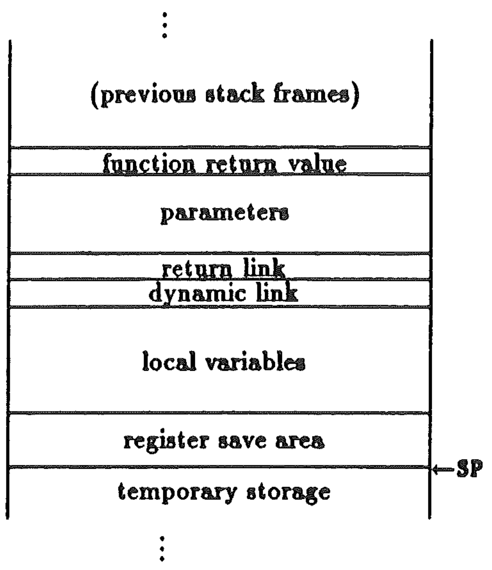


Figure 2-2. Format of a Stack Frame.

Not all of the fields will be used by the compiler for every procedure; only the return link is present in every frame. It is the responsibility of the called procedure to remove the parameters and local variables from the stack before a return is made to the caller.

Function Return Value The function return value field appears only for functions. ~~A~~ value assigned to the function name within the function is stored in this location and left on the top of the stack when the function returns. Space for this field is allocated by the calling routine before evaluation of the arguments for the function call. The space is removed from the stack when the calling routine has no further use for the value.

Parameters The parameter area has an entry for each parameter to the procedure or function. The entry for a value parameter contains the value of the corresponding argument, while the entry for a variable parameter contains the address of the argument. Parameters are pushed onto the stack as they are evaluated, in left to right order, so the first parameter to a procedure is the first one pushed onto the stack.

Return Link The return link is the address to which control is transferred on return from the procedure or function.

Dynamic Link By default, procedures compiled with **walkback** enabled establish a dynamic link that points to the dynamic link in the previous stack frame. The base of the linked list of stack frames is contained in the first word of the pssct **P$DYNL**. When a run-time error is detected, the dynamic link is used to show the procedure calls that led to the error (the procedure walkback). A dynamic link is not present in the stack frame for procedures compiled with the command line switch **nowalkback**.

Local Variables The local variable field contains space for all local variables of the procedure or function. The field is allocated upon entry to the block.

Register Save Area This area saves the values of all registers used within the procedure. The registers are saved on entry to the procedure and restored on exit. Only registers actually used are saved. The general registers are stored first, with the highest register used pushed first. (This is important to the algorithm for locating variables in lexically enclosing blocks.)

Temporary Storage In the process of generating code, expressions that are used more than once are computed and the values saved. These values may be saved on the stack if no register is available to hold them. Also, the stack is used to interface with support library routines and the operating system.

Monitoring Memory Usage

A Pascal task is typically arranged in memory with the program code written to low memory, followed by the default allotment of 4K bytes (2K words) for the stack. (In this discussion, a typical Pascal task uses no Task Builder options to rearrange memory.) The remaining memory is unallocated and available for the heap.

To arrive at this arrangement, the following events occur:

1. The program code is loaded into memory, with the stack in low memory.
2. The support library initialization procedure is called, initializing the support library and performing the next three steps. (See the "Support Library" section.)
3. The program is expanded by 4K bytes to make room for the stack.
4. The stack is moved from low memory to its new location immediately following the program code. In this case, the program section **$$HEAP** has a default length of zero.
5. The user program is started. During execution, if the heap is exhausted, memory is taken from the stack area.

However, this arrangement changes if memory is explicitly allocated to **$$HEAP**. Using the **EXTSCT** Task Builder option, you can set the size of **$$HEAP** to any value up to the maximum size for the particular program. (When **$$HEAP** is set to its maximum, it includes all memory not allocated to the program code.) When **EXTSCT** is used, the sequence of events leading to execution is:

1. The program code is loaded into memory.
2. The support library initialization procedure is called to initialize the support library and perform the next three steps.
3. The stack is moved from low memory to the top of the **$$HEAP** psect.
4. The maximum stack size is set to the size of **$$HEAP**.
5. The user program is started.

The remaining memory is available for heap expansion. Again, if the space used for the heap is exhausted, the heap expands into the stack.

When the size of **$$HEAP** is set to its maximum for a given program, the heap is no longer expandable. In this situation, the stack and heap "share" the same memory, meaning the stack begins at the high end of memory and grows down, and the heap begins at the low end and grows up.

In certain applications you may find it advantageous to keep track of **$$HEAP** memory as it is allocated to a program. The Pascal support library contains three routines—**space**, **p$inew** and **p$dispose**—that allow you to keep track of memory allocated to **$$HEAP** (either the stack, or the stack and heap). Briefly, the **space** function returns the amount of **$$HEAP** (stack, or stack and heap) space available to an executing program at a particular time. The **p$inew** function returns the address of a block of memory having a specified length. The **p$dispose** procedure deallocates blocks of memory allocated by **p$inew**. In a later subsection, an example of the boolean function **NewOK** is provided, which not only shows the correct way to use the three routines but also is useful in determining whether enough memory is available to satisfy a request for a block of memory.

Two reasons for monitoring the size of **$$HEAP** are to find out how close the program is to running out of memory and to find out whether enough memory is available to perform a given subtask. For example, a checkers-playing program could use these functions (as in **NewOK**) to determine the number of moves that the program can look ahead based on the amount of memory available to perform the look-ahead.

Space can be called independently, whereas **p$dispose** must be used to deallocate memory allocated by **p$inew**. The routines are described in detail below.

CAUTION

Very large stack overflows, those that extend beyond the heap and into the program code sections of memory, have the potential for causing serious problems. In some cases it is possible for the error to prevent the appropriate error message from being printed or the program from properly terminating. In some rare instances, the condition can even lead to the disruption of the computer's operating system. It is the programmer's responsibility to avoid such excessively large overflows by controlling the size of local variables and value parameters passed to procedures.

The 'Space' Function

The **space** function is used to determine the amount of stack and **$$HEAP** space available to an executing program. With this function you can determine how close the program is to running out of memory.

The function **space** must be declared **external** to the program, as shown:

function Space: *functype*; **external;**

where *functype* is the data type of the function and its returned value. The function is usually declared of type **integer** but another data type similar to **integer**, "unsigned" (0..65535), can be used to represent the value. The value returned by space depends on the arrangement of the stack and heap and the size of $$HEAP.

Figures 2-3, 2-4 and 2-5 on the next several pages are memory diagrams showing a task's arrangement in memory based on the size of the $$HEAP psect (set at task-build time). The diagrams also show the relationship between the size of $$HEAP and the returned value of the space function. These diagrams are designed as an aid in visualizing the use of the stack and heap and as an aid in understanding the way the value of space is arrived at.

The figures are divided into "times," or snapshots of a program in memory at various stages of its execution. Time 0 in the three figures is the point at which the user program actually begins execution, after the program code is loaded into memory, the support library is initialized and the stack is moved.

The figures contain the following five symbols. Curled braces designate the value that the space function would return at the indicated time. *M* signifies the total amount of memory being used by the program at the indicated time. Vertical arrows represent stack and heap expansion. SP signifies the stack pointer. 64KB stands for 64K bytes or 32K words, the high end of memory; 0KB denotes the beginning of memory.

Figure 2-3 illustrates the memory arrangement of a program task-built with no options. By default, the size of $$HEAP is zero and the size of the stack is 4KB. With this arrangement, the space function monitors the use of the stack only and has no way of knowing how much heap space has been used or how much remains.

In Figure 2-4 we see a similar layout, except that the $$HEAP psect is extended to a specific value with the EXTSCT Task Builder option. The length of $$HEAP can be any value up to the maximum for a particular program and can be less than the default (4KB) if you want to conserve space. As with the previous arrangement, the space function monitors the use of the stack only and has no way of knowing how much heap space remains.

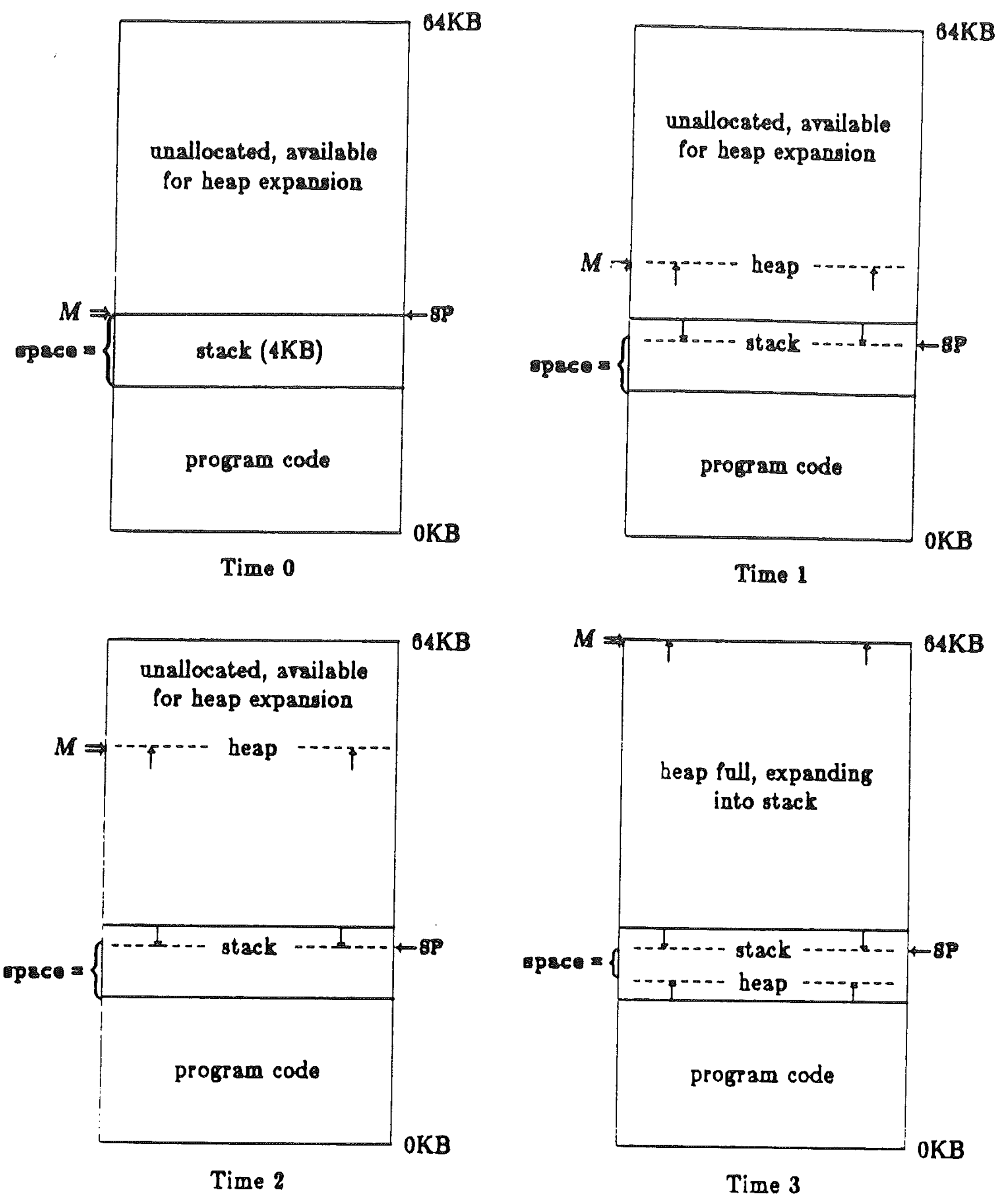


Figure 2-3. Memory arrangement of a program task-built with no options.

At Time 0, the support library has been initialized but no Pascal statements have been executed. The size of the stack is 4KB (2K words) and the size of **$$HEAP** *is zero. As time progresses, the heap and 'M' (the amount of memory used) grow toward 64KB (32K words) and the value of* **space** *decreases as the stack grows downward. At Time 3 (should the program run that long), memory available for heap expansion is exhausted. The heap then expands into the stack area, further reducing the value of* **space**.

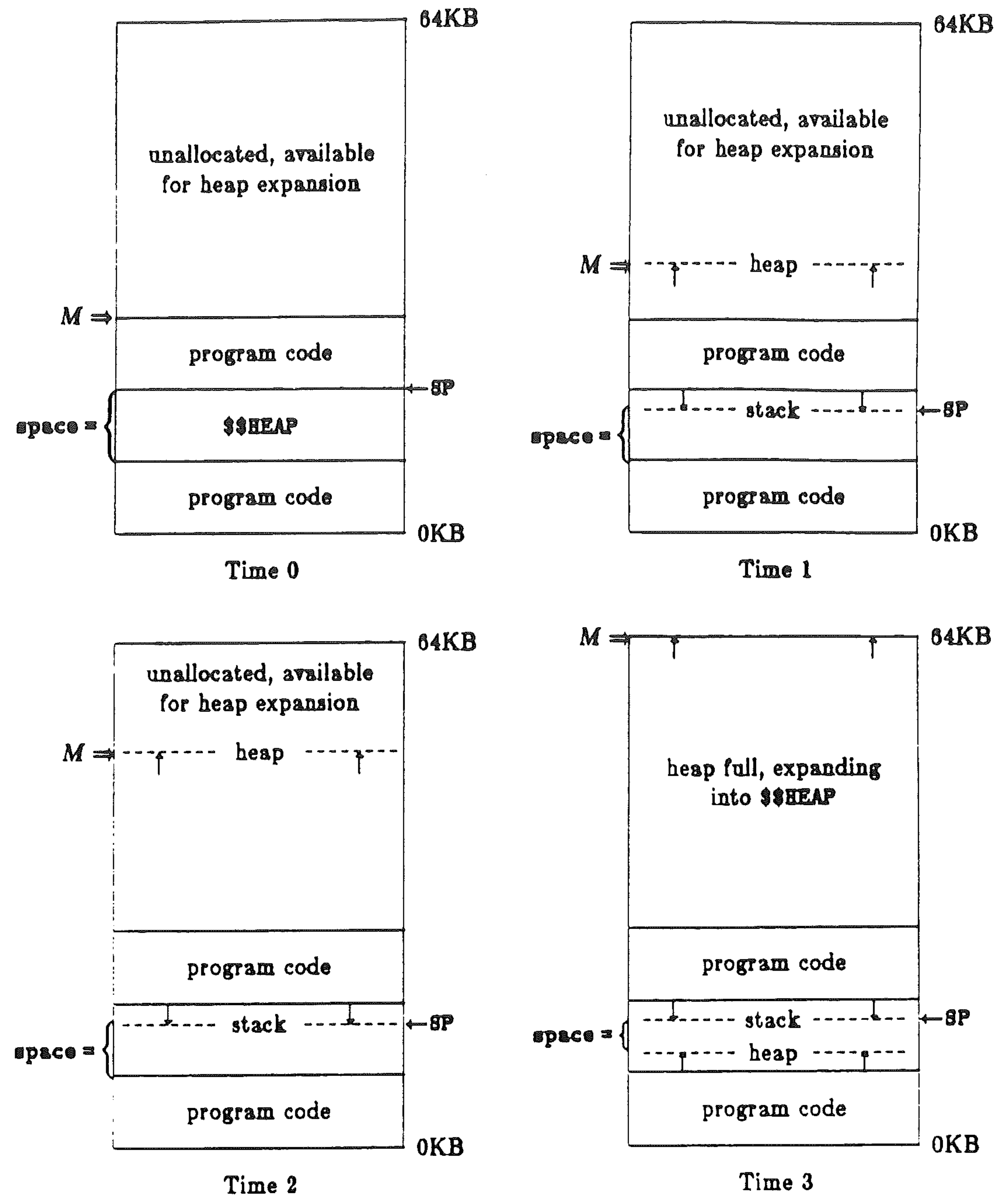


Figure 2-4. Memory arrangement of a program task-built with EXTSCT=$$HEAP:n option.

The stack and the psect **$$HEAP** *share the same memory area. The Task Builder splits the program code into two sections with space for* **$$HEAP** *allocated between the code sections. At Time 0, the support library has been initialized but no Pascal statements have been executed. As time progresses, the heap and 'M' grow toward 64KB (32K) and the value of* **space** *decreases as the stack grows downward. At Time 3 (should the program run that long), memory available for heap expansion is exhausted. The heap then expands into the stack area, further reducing the value of* **space**.

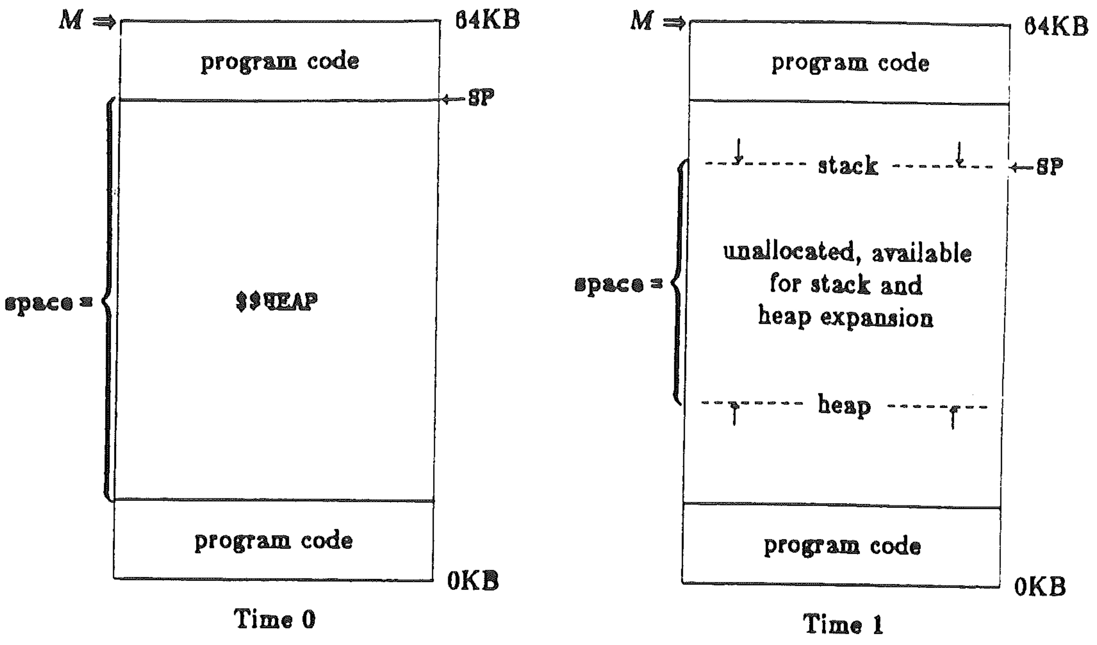


Figure 2-5. Memory arrangement of a program task-built with $$HEAP set to its maximum.

The size of $$HEAP is set to include all remaining memory not allocated to the program code. (See text below for the way to do this.) In this situation, **space** *returns the amount of $$HEAP space available for use by the program. At Time 0, the support library has been initialised but no Pascal statements have been executed. Note that unlike the two previous figures, $$HEAP contains both the stack and heap, which grow toward each other and ideally never meet. If* **space** *is called at Time 0, the value it returns is equal to the size of $$HEAP. At Time 1, the amount of $$HEAP remaining and the value of* **space***, being identical, both decrease by the same amount as stack and heap space is allocated. As time progresses, memory available for the stack and heap continues to be used and disposed of until the program ends or until $$HEAP is exhausted and the program aborts.*

To use the **space** function to the greatest advantage, set the size of $$HEAP to the total amount of unallocated memory. As Figure 2-5, Time 1, shows, the stack and heap share the same block of memory. With this arrangement the stack begins at the high end of $$HEAP and grows down; the heap begins at the low end of $$HEAP and grows up. Here the **space** function measures the amount of unallocated memory in $$HEAP, providing a true reflection of the amount of memory available to the program.

To set the length of $$HEAP to its maximum value, follow these five steps. (The program used in this short tutorial is ENTRY.PAS.)

1. Compile the program normally.

2. Once it has compiled, task-build it with no options to produce a load map file. The first file name specifies the name of the map file. For ENTRY.PAS, the command is:

```
>TKB ENTRY/FP/CP,ENTRY=ENTRY,LB:[1,1]PASLIB/LB
```

3. Examine the map file (ENTRY.MAP) for the program's maximum task address limit (in other words, the length of the task). The "task address limits" line is near the beginning of the map file and looks like this:

```
Task address limits: 000000 137673
```

   The second number, 137673 above, is the value you need. This number is in octal bytes.

4. Subtract the maximum task address limit from the total amount of memory available to a program (177700_8 bytes). Remember, the subtraction is performed in octal. The result is the value you enter on the **EXTSCT** option in the second and final task-build. In the example the result is 40005_8, which can be rounded off to 40000_8.

5. Run the Task Builder to build the program a second time. This step creates the desired executable task arranged in the same layout as Figure 2-5. The command line for the second build is:

```
>TKB
TKB>ENTRY/FP/CP,ENTRY=ENTRY,LB:[1,1]PASLIB/LB
TKB>/
Enter options:
TKB>EXTSCT=$$HEAP:40000
TKB>//
```

5. (This step is optional.) Check the second map file for a task size somewhere near 32700 words and a maximum task address limit somewhere near 177700_8 bytes. If the figures you see are reasonably close to these figures (within 100_8 bytes), you have correctly set the length of **$$HEAP** to the amount of unallocated memory. The task is now ready to be executed.

Be aware of the fact that the space function does not account for the fragmentation of the heap as a result of calls to **new** and **dispose** and the opening and closing of files. Unused portions between the low and high end of the heap are treated as if they are allocated.

See the example under "Example: Function NewOK" for use of **space**.

For additional information on the **EXTSCT** option refer to "The Task Builder" section in this guide.

Function 'P$inew' and Procedure 'P$dispose'

The function **p$inew** and procedure **p$dispose** are entry points for the standard procedures **new** and **dispose**. **P$inew** allocates a specific sized block of memory, and **p$dispose** deallocates a specific block. As a comparison, the standard procedures **new** and **dispose** determine the size of the block to allocate or deallocate from the length of the data type.

A cache system is an example of a situation in which you might use **p$inew** and **p$dispose** procedures. The program needs to use as much memory as is available for data storage before it writes the data to a file.

Defined in the support library, these two routines must be declared **external** to the program, as shown:

function P$inew(*blocksize*: *argtype*): *functype*; **external**;

procedure P$dispose(*pointer*, *blocksize*: *argtype*); **external**;

where

blocksize is the size of the memory block to be allocated or deallocated, in bytes.

pointer is the address of block to deallocate.

argtype is the data type describing *size* and *pointer*.

functype is the data type of the function's return value. The returned value is the address of the block. If there is not enough contiguous memory available to satisfy the request, a value of 0 is returned for numeric data types, **nil** for pointer types. The most common types used with **p$inew** and **p$dispose** are the standard type **integer** and a user-defined type **unsigned**, in the range 0..65535.

For an example showing the use of **p$inew** and **p$dispose** see the code for procedure **NewOK**, below.

Example: Function 'NewOK'

The boolean function **NewOK**, provided below as an external, uses the three routines previously described functions to determine whether a block of memory can be allocated, leaving a specified amount of stack space. We recommend that you reserve 200_8 to 1000_8 bytes of stack space for parameter and local variable storage and for error processing. The amount of memory you reserve depends on the parameter and local variable requirements of the procedures being called by the program. For instance, if the program calls numerous procedures, each containing a large number of parameters and local variables, the amount of memory to reserve would be greater than for a program that uses smaller procedures.

NOTE

If a program that uses NewOK aborts with a "stack overflow" error, the amount of reserved memory is probably not large enough for the amount of stack space required for parameters and local variables. To alleviate this error, increase the amount of memory you are reserving for the stack.

In addition to showing the correct way to use the three routines, NewOK can be incorporated into your programs when you need to determine if a request for memory will fail.

NewOK, which could be stored in NEWOK.PAS, returns a `true` value if the block can be allocated, `false` if the block cannot be allocated. The first argument to NewOK is the size, in bytes, of the memory to be allocated. Use the `size` function to determine the size of a Pascal data type. (The `size` function is described in the Language Specification.) The second argument is the amount of stack space in bytes that you want to remain unallocated, if the block is allocated.

First, the function allocates the desired block of memory with the call to `p$inew`. If `p$inew` returns a zero, this means that there was not enough memory available to allocate a block of that size, and NewOK returns `false`. However, just because the block was allocated does not necessarily mean that NewOK returns `true`. If the returned value of `space` is less than the amount of stack space you have reserved for variables, etc., NewOK is set to `false` because memory would be taken from the reserved block. Of course, the function returns a `true` value if the block was safely available. Finally, the same block of memory is disposed of by `p$dispose`. (Remember, NewOK only checks to see if a block of memory could be allocated. To allocate the block, call `new`.)

Example: Routines For Determining Memory Available (NEWOK.PAS)

```
{$nomain}
type
  Unsigned = 0..65535; { Unsigned integer }

function p$inew(Blocksize: Unsigned): Unsigned;
  external;      { Allocate a specific sized block of memory }

procedure p$dispose(Pointer, Blocksize: Unsigned);
  external;      { Dispose of a specific sized block of memory }

function space: Unsigned;
  external;      { Determine amount of stack ($$HEAP) space left }

function Newok(Reserved, Stackspace: Unsigned): boolean;  external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

function NewOk;

  var
    P: Unsigned;  { Address of block if allocated }

  begin  { NewOK }
    P := p$inew(Reserved);              { Try to allocate block }
    if P = 0 then NewOK := false        { No luck }
    else
      begin
        NewOK := space >= Stackspace;   { Check for enough stack left }
        p$dispose(P, Reserved);         { Deallocate the block }
      end;
  end;  { NewOK }
```

To show the correct way to set up the critical variables and call NewOK, the sample program CHECKTEST uses NewOK to find out how many moves it can look ahead. For illustrative purposes, the program, CHECKT.PAS, simulates a real checkers program, making use of a 20,000-word array Dum to produce a larger task size. CHECKT uses a linked list to store the look-ahead moves a normal checkers program would make. The program merely illustrates the use of NewOK to perform the look-ahead.

Example: Program to Play Checkers (CHECKTEST.PAS)

```
program CheckTest;

const
  Reserved = 200; { amount of stack space to reserve }

type
  Unsigned = 0..65535;
  Ptr = ^Node;                  { Pointers into search tree }
  Node = record                 { Node in search tree }
    Father: Ptr;                { Father of this node }
    Son: Ptr;                   { Pointer to best son }
    Brother: Ptr;               { Link to next brother }
    Value: integer;             { Value of this board position }
    Move: integer;              { Move descriptor to reach node }
    Jump1, Jump2: integer;      { Jumped pieces removed by move }
    Mobility: integer;          { Mobil and deny }
    Attack: integer;            { Pin and threat }
    Gradient: integer;          { Target gradient }
    Bits: integer;              { Scoring bits }
    end;

var
  Alloc: boolean;
  P: Ptr;
  Movesize: unsigned;
  NumMoves: unsigned;                  { number of moves }
  Nextmove: node;
  Dum: array [1..20000] of integer;  { Dummy array simulating a long program }

function NewOK(Reserved, Stackspace: Unsigned): boolean;
  external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

begin  { CheckTest }
  Movesize := size(Node);
  writeln('The size of a move is: ',Movesize:1);
  new(P);
  NumMoves := 1;
  repeat
    alloc := Newok(Movesize,Reserved);
    if alloc then begin
      new(P^.son);
      P := P^.son;
      NumMoves := NumMoves + 1
      end;
  until not alloc;
  write('The number of moves necessary to fill up the heap is: ');
  writeln(NumMoves:1);
end.  { CheckTest }
```

The compilation process for this program is as follows:

```
>PAS NEWOK ———————————————————— nomain is embedded
>PAS CHECKT
>TKB CHECKT/FP/CP,CHECKT=CHECKT,NEWOK,LB:[1,1]PASLIB/LB
>TKB
TKB>CHECKT/FP/CP,CHECKT=CHECKT,NEWOK,LB:[1,1]PASLIB/LB
TKB>/
Enter options:
TKB>EXTSCT=$$HEAP:37765 ————————————— value from first TKB map file
TKB>//
>RUN CHECKT
The size of a move is: 22
The number of moves necessary to fill up the heap is: 724
```

Storage Allocation

The compiler assigns storage for variables of pre-declared types as shown in this table:

Type	Size (bytes)	Alignment (bytes)
Boolean	1	1
Char	1	1
Integer	2	2
Real	4	2* ($double off)
Real	8	2* ($double on)
Text	2	2

Space for user-defined types is allocated as follows:

Enumeration If the type has up to 256 members, it is allocated one byte aligned on a byte boundary. If it has more than 256 members, it is allocated two bytes, aligned on a two-byte boundary.

Subrange Allocated in the same way as the parent type. Note, however, that a range such as 0..255 or −127..127 is stored in two bytes except inside of a packed structure, where it is stored in one byte. An unsigned number (0..65535) is stored in two bytes.

Pointer Allocated two bytes, aligned on a two-byte boundary.

Array Allocated the amount of space needed to hold the number of elements specified, aligned in the same way as the element type. The elements are placed in ascending memory locations.

Set Allocated one bit for each member of the base type, with the total size rounded up to the next larger full byte. Bit allocation begins with the least significant bit of the first byte. If the size is a single byte, it is aligned on a byte boundary; otherwise it is aligned on a two-byte boundary. A base type that is a subrange is expanded to the full range of possible values before the set is allocated. For example:

```
type
  Color = (Red, Orange, Yellow, Green, Blue);
  Hot = Red..Yellow;

  Colorset = set of Color;
  Hotset = set of Hot;
```

In this example, Hotset is allocated the same amount of space as Colorset. A maximum of 256 members is allowed; a base type of integer, or any integer subrange, has members from 0 to 255.

Record Each field in the record is allocated space in the same way as a variable of the same type, in the order specified. The alignment of the record is the maximum of the alignments of its fields.

`Packed Array` The number of bits needed to contain each element is computed. For example, the subrange type `0..3` requires two bits to contain a value. If the space required for an element is less than a word, the element size is increased to the smallest power of two bits (1, 2, 4, 8, 16) that contains the value. The array is allocated space to hold the number of elements specified, where each element is considered to be of the size just computed. If elements are allocated eight bits or less, the array is aligned on a byte boundary. If the elements require a word or more, space is allocated as for a normal array type.

`Packed Set` The same as unpacked sets, except that the size is not rounded up to an even byte and the alignment is to a byte boundary.

`Packed Record` Each field in the record is allocated exactly the number of bits required to contain it, except that a field of a simple type that would span or cross a word boundary is forced to begin at a word boundary. Fields are allocated in the order declared, beginning at bit zero (least significant bit).

NOTE

The predefined functions `size` and `bitsize` report the amount of storage allocated to any user-defined structured type. See the Language Specification for details.

Run-Time Error Reporting

The Pascal-2 run-time error reporting system is intended to simplify error analysis by reporting run-time errors in terms of source lines and procedure names. Upon detecting a run-time error, the reporting system prints a short description of the error, then traces the execution history, procedure by procedure, from the point of error back to the main program. This is called a "walkback," or "traceback."

Errors are detected by the hardware or by special checks inserted in the generated code. After an error is detected, control of the program then passes to an error routine, which closes all open files, then prints an error message and stack traceback.

The walkback consists of the following:

- The message header, which includes the task name, type of error, and program counter at the time of the error. The task name may be either an installed task name or the terminal port from which the program was executed with the **RUN** command. The two types of run-time errors are "fatal" and "I/O." Fatal errors are unrecoverable; I/O errors are recoverable. For details on I/O-error recovery, see "I/O Error Trapping" later in this section. The program counter is the location at which the error occurred. If you utilize the walkback, the program counter is of little use to you since the location of the error is given as a line number in a procedure.

- A description of the error. See Appendix B of this section for a detailed explanation of the error messages. By modifying OP-ERRO.PAS, you can change the wording of Pascal run-time messages if you so desire. However, you cannot change the text of RSX system I/O error messages. See the section on "Customizing Error Reporting" for details.

- For I/O errors, the error code and the file name of the file causing the error. The error code is printed in both decimal and octal. On RSX, all I/O errors have negative error codes. These error codes are FCS errors detected by the system and are described in the *RSX I/O Operations Reference Manual.* The file name includes the device name, file name, extension, and version number but not the UIC of the file, because it is not available to the support library.

- The location of the error in terms of line number and procedure name. The line number refers to lines in the overall source program, not statements in individual procedures. (For external procedures, the line number refers to lines in the external module.) A special case arises when a run-time error is detected in an external procedure compiled with **nowalkback**. In this case, the location of the error is given as an octal address.

- The reverse sequence of active procedure calls back to the main program, if the error occurred at a level other than the main program.

The walkback may be disabled at compile time; to do this, use the **nowalkback** compilation switch. When **nowalkback** is selected, the message header and the error message are printed but not the walkback.

WARNING

Error walkback cannot be used with Instruction & Data (I & D) space programs available on RSX-11M-PLUS. Normally (in non-I & D space programs), the walkback routine examines instructions to find special markers and pointers to data areas, which contain procedure names and statement locations. But, in I & D space programs, instructions and data are kept separate, and this, coupled with the fact that data sections can be overlaid, makes it difficult for the walkback routine to function properly. In some I & D space programs, the walkback routine reports addresses instead of procedure names, but in other instances, the walkback procedure can abort with an odd address error or memory management trap. You should generally compile all I & D space programs with the **/nowalkback** switch to avoid such problems.

Example 1

This example provides a look at a possible run-time error condition and the resulting error walkback.

```
>RUN PACKER

TT0 -- Fatal error at user PC= 2316
Array subscript out of bounds

Error occurred at line 140 in procedure arrangetree
Last called from line 326 in procedure switchnodes
Last called from line 402 in procedure getdep
Last called from line 579 in program packer
```

Example 2

A procedure called recursively may have many consecutive activations. In this case, the number of identical lines is indicated by the note (nn times) after the location description. Appearing below is the walkback of a program that looped recursively until the stack overflowed.

```
>RUN EXTRA

TT1 -- Fatal error at user PC= 5223
Stack overflow. Try expanding $$HEAP

Error occurred at line 124 in procedure walk
Last called from line 290 in procedure reanalysis (898 times)
Last called from line 423 in procedure unloadbits
Last called from line 440 in procedure matrixmask
Last called from line 535 in procedure processleftop
Last called from line 608 in program extra
```

Example 3

This example illustrates the walkback produced as a result of an I/O error that is not trapped by the user. The program is an installed task.

```
>FMT
File to reformat: LSAT.TXT

FMT -- I/O error at user PC=2166
Can't open file
I/O error code= -26. (346B) in file: DL0:LSAT.TXT;0

Error occurred at line 44 in procedure openfile
Last called from line 69 in program format
```

Example 4

This example shows the walkback produced as a result of a run-time error in an external procedure compiled with **nowalkback**. Note that if the external procedure is compiled with **walkback** (the default), the location of the error is in source terms.

```
>RUN DIFFS

TT2 -- Fatal error at user PC= 1336
Division by zero

Error occurred at location   1336
Last called from line 32 in program diffs
```

I/O Error Trapping

Pascal-2 permits you to write programs that trap and detect many kinds of I/O-related errors that normally would be fatal. Three predefined routines — procedure **noioerror** and functions **ioerror** and **iostatus** — facilitate this trapping of I/O errors. Using these routines, you have the ability to process I/O errors with your own code. You have two options: terminate the program at the occurrence of an I/O error (you can print your own diagnostics), or continue execution in spite of the error. The choice depends on the need.

Since these three routines are predefined in the compiler, they do not need to be declared in your program. They accept a file variable as their only parameter. Details are supplied below.

procedure **noioerror**: Specifies that the calling program will handle any I/O errors that result from reading or writing to the specified file. The file must be open before **noioerror** is called.

function **ioerror**: Determines the status of the last I/O operation that the program performed on the specified file. This **boolean** function returns a **true** value if an I/O error has occurred, **false** if the operation was successful.

function **iostatus**: Returns the integer error code that describes the last attempt to access ~~a~~ file. This function helps your program determine the cause of the error. Your program can either bypass the problem and continue processing or terminate so you can correct the problem.

I/O error codes can be negative or positive. A negative error code (negated by the Pascal-2 support library) indicates that the error is RSX -specific. RSX I/O error codes are listed in the *RSX I/O Operations Reference Manual.* A positive error code indicates that the error was detected by the support library. Pascal-2 error codes, along with the text of the error message and a brief explanation of the cause, are listed in Appendix B of this reference.

If you pass **iostatus** as a parameter to the external procedure **sayerr**, **sayerr** prints the text of the error message corresponding to the returned value of **iostatus** (see "Procedure SayErr" below).

When you call these routines, you are responsible for checking the status of each I/O operation, to ensure that it was successful. If you fail to check the status and an error occurred, the results are unpredictable.

The Pascal-2 support library places information about run-time errors into registers. Sometimes a program may fail with a run-time error and the the library encounters another error while trying to report the first one. In such cases, the error reporting system prints the message "Multiple errors detected" then the operating system prints a message "Unexpected trap" and a register dump. For the original error, the PC address is in register R0 and the Pascal error code is in R1. Registers R3 and R4 contain the PC address and Pascal error code for the second error. To find the error, you must get your information from the register dump.

The following program illustrates the use of these procedures. The program is designed to continue executing despite an I/O error. The call to noioerror indicates to the run-time system that the program handles errors detected on the standard file input.

Example: Use of I/O Error Trapping Routines

```
program Iotest;

  var
    I, Times: integer;

  begin
    Noioerror(input);
    for Times := 1 to 4 do
      begin
        write('Type an integer: ');
        read(I);
        if Ioerror(input)
          then writeln('Error detected.  Status=', Iostatus(input))
        else writeln('The integer was: ', I: 1);
        readln;
        writeln;
      end;
  end.
```

If this program is compiled and run, the following results might be produced. The first entry results in a successful read of the integer I. The second and third entries result in a Pascal-2 run-time error.

See Appendix B for a list of run-time error messages and associated error codes. The final entry is successfully read, and the program ends. (Under normal conditions, the first error would cause the program to abort.)

```
>RUN IOTEST
Type an integer: 1234
The integer was: 1234

Type an integer: 123456789 ——————————— integer too large
Error detected.  Status=     19———— Pascal–2 error code for invalid integer

Type an integer: FFG ———————————— non-integer characters
Error detected.  Status=     19———— Pascal–2 error code for invalid integer

Type an integer: 77
The integer was: 77
```

The I/O error-trapping procedures can be used to determine the reason that a file could not be opened. To use this feature, specify the fourth parameter on calls to **reset** and **rewrite**. Specifying this fourth parameter keeps the **reset** or **rewrite** from trapping a normally fatal "open" error. This allows your program to recover and continue or terminate under your control.

The sample program OPNERR illustrates the use of **ioerror** and **reset/rewrite**. The program attempts to open a file called TEST.DAT on the device **XXXX:**, a fictitious device name. The error is detected by **Ioerror**.

```
program Opnerr;

  var
    F: text;
    Status: integer;

  begin
    reset(F, 'XXXX:', 'test.dat', Status);
    if Ioerror(F) then
      writeln('I/O status=', Iostatus(F));
  end.
```

When this program is compiled and executed on RSX, it yields the following output. The value -55 is the RSX I/O error code for "bad device name." To print the text of the error message, use the **sayerr** procedure, described below.

```
>RUN OPNERR
I/O status=    -55
```

Procedure 'SayErr'

The text of a given I/O error message can be printed if you call the external procedure **sayerr**, which is supplied in the Pascal support library. **Sayerr** is not pre-declared, so you must declare it when you use it, as shown:

```
procedure SayErr(status: integer);  external;
```

where *status* is the error code of the error.

Procedure **sayerr** takes an RSX I/O error code (a negative number) and looks in the system file `LB:[1,2]QIOSYM.MSG` to print the text of the error message. **Sayerr** will only print messages for negative I/O error codes in the range $-255..-1$; **sayerr** ignores error codes that lie outside this range, printing nothing.

The program listed above can be modified to call **sayerr** to print the text of the error message corresponding to RSX error number -55, as shown below. The integer function **iostatus** is the parameter being passed to **sayerr**. The parameter takes on the value returned by **iostatus**, in this case -55.

```
program Opnerr;

  var
    F: text;
    Status: integer;

  procedure SayErr(Code: integer);
    external;

  begin
    reset(F, 'XXXX:', 'test.dat', Status);
    if Ioerror(F) then
      begin
        writeln('I/O status=', Iostatus(F));
        SayErr(Iostatus(F));
      end;
  end.
```

When this program is compiled and run on RSX, it yields these results:

```
>RUN OPNERR
I/O status=    -55
Bad device name———————— error message printed by sayerr
```

Customizing Error Reporting

The flexibility of the Pascal–1 and Pascal–2 run-time systems allow Pascal–2 run-time organization allows you to not only handle I/O errors in your code but to customize the run-time error diagnostics to suit your needs. Included in the distribution kit are two Pascal source files, OPERRO.PAS and UERROR.PAS, which let you modify the way in which errors are reported. The object-file equivalents of these two procedures are in the Pascal support library. The changes you make can be used for a one-time debugging run, or they can be permanently installed in the support library.

OPERRO.PAS contains the entry point **P$ERROR**, which the support library's error-handling routine calls to print the message header and text of the run-time error message. This source file is provided so you can change the wording of any error message simply by editing the source.

UERROR.PAS contains the entry point **P$UERROR**, which is called following **P$ERROR** to print additional information about the error. This procedure contains three boolean constants set to **false** in the release version, each controlling (inhibiting) the printing of a separate set of diagnostics. Initially, with the booleans set to **false** nothing is printed. But by editing UERROR.PAS and setting one or more constants to **true**, you can receive a file dump of the offending file, a memory map of the program, and/or a brief message describing the error. The **const** fragment below shows the three constants.

```
const
  DumpMemory = false; { Print a memory map }
  DumpFile = false; { Print detailed file dump }
  PrintErrorText = false; { Print text of I/O errors }
```

By using UERROR.PAS, you can add your own code to UERROR.PAS for the printing of more specialized debugging information, and you can print out the values of critical variables used in the program. To print variables you must include the program's global variable declarations in UERROR.PAS. The ability to print critical variables is useful when you have a program with many overlays or when the program is too large to run with the Debugger.

When a run-time error is detected, several steps are taken:

1. Control of the program transfers to the **$ERR** error-control module, in the Pascal support library. **$ERR** collects information about the error from the library data area.

2. **$ERR** calls **P$ERROR** (in OPERRO.PAS) to print the message header followed by the error message.

3. **$ERR** then calls **P$UERROR** (in UERROR. PAS), which does nothing (by default) but can be modified to print a more detailed error message, dump the contents of the offending file, and/or print a memory map of your program.

4. On return from **P$UERROR**, the error-control routine **$ERR** transfers control to the Post-Mortem Analyzer (PMA), which prints the error walkback, and the program terminates.

When you use a modified version of one or both of these procedures as externals, you need not declare them explicitly in the main program. After the altered version(s) of these procedures are compiled (with **nomain**), simply specify the module name(s) on the Task Builder command line after the program name. The Task Builder substitutes your version for the version in the support library. This sequence of commands should be used.

```
>PAS PROG
>PAS UERROR/NOMAIN
>PAS OPERRO/NOMAIN
>TKB PROG/FP/CP,PROG=PROG,OPERRO,UERROR,LB:[1,1]PASLIB/LB
```

Another way to override the version in the support library is to include the modified OPERRO.PAS and/or UERROR.PAS as part of the main program. The **%include** directive does this easily. (See "Implementation Notes" in this section for use of **%include**.)

For example:

```
%include 'operro';
%include 'uerror';
```

When using the **%include** directive, compile and task-build the main program as you normally would. The Task Builder resolves the references to **P$ERROR** and **P$UERROR** with the procedures included in the program. The program **PROG**, above, would be task-built in this manner with these commands:

```
>PAS PROG
>TKB PROG/FP/CP,PROG=PROG,LB:[1,1]PASLIB/LB
```

In UERROR.PAS, if the constant **PrintErrorText** is set to **true**, the support library procedure **sayerr** is called and the text of an I/O error message is printed based on an integer error code. **Sayerr** is described in the section on "I/O Error Trapping."

The constant **DumpMemory**, if set to **true**, causes the program to print a memory dump, or map, showing the program's use of memory. The map is printed by the external procedure **memmap**, a Pascal support library routine. Although **memmap** is declared in UERROR.PAS, you can use it with any Pascal program, independent of UERROR.PAS. Simply declare it in the program as an external with no parameters. The map helps you determine the way dynamic memory is being allocated and perhaps the reason your program is running out of memory.

Example: Memory Map Procedure's – OUTPUT

```
>RUN DIAL

TTO -- Fatal error at user PC= 1276
Attempted reference through NIL pointer

Memory map:

 Start      End    Size       Description

     0 -    171  (122.)      Low memory
   172 -  16567  (7422.)     User program code
 16570 -  16577  (8.)        Global level data
 16600 -  35303  (7492.)     Pascal support library code
 35304 -  53605  (7362.)     System library code (SYSLIB.OLB)
 53606 -  54011  (132.)      Library data area
 54012 -  56677  (1462.)     Used
 56700 -  66163  (3764.)     Stack space available
 66164 -  66677  (332.)      Stack space used
 66700 -  66715  (14.)       Active file table
 66716 -  66745  (24.)       File variable for file: TI:
 66746 -  67065  (80.)       Record Buffer for file: TI:
 67066 -  67115  (24.)       File variable for file: TI:
 67116 -  67235  (80.)       Record Buffer for file: TI:
 67236 -  67425  (120.)      Used
 67426 -  67455  (24.)       File variable for file: DB0:REFER.TMP;0
 67456 -  67615  (96.)       File descriptor block for file: DB0:REFER.TMP;0
 67616 -  67671  (44.)       Free
 67672 -  67701  (8.)        Used
 67702 -  70111  (136.)      Free
 70112 -  71337  (662.)      Used
 71340 -  73677  (1248.)     Free
 73700 - 177677  (34816.)    Available memory not used by task

Task size=30656. bytes

Error occurred at line 55 in program dialphones
```

The constant `DumpFile` can be used to print a detailed dump of the Pascal and RSX file structures when an I/O error is detected. When `DumpFile` is set to `true`, the support library module `fdump` is called to dump information about the file. Programmers familiar with the structure of an FCS File Descriptor ~~Block~~ (FDB) may find this information useful in diagnosing obscure file problems.

The following file dump is only partially listed because of the length of the `fdump` listing. Enough information is provided to give you an idea of the usefulness of the file dump. Although you wouldn't want the file dump printed at each occurrence of a run-time I/O error, in certain circumstances the file dump may help you diagnose more obscure errors.

Example: Detailed Dump of Pascal and RSX File Structures

```
>CPR
First file: DISPLA.PAS
Second file: NEWDIS.PAS

CPR -- I/O error at user PC= 1276
Can't open file
I/O error code= -26. (346B) in file: DR0:NEWDIS.PAS;0

File information for file variable at: 101626B

Contents of file variable:

Ptr:       0      Pointer to data in file buffer
Lun:       2      Logical Unit Number
Stat:     24B     File status
                    Current character not defined
                    Text file
                    Input operations permitted
Iov:    31102B    I/O vector address

                  :
                  :
                  :----------------- more information about file variable

Contents of File Descriptor Block:

Rtyp:      2      Record type
                    Variable length records
Ratt:      2      Record attributes
                    Normal carriage control
                    Records cross block boundaries
Rsiz:     132.    Record size

                  :
                  :
                  :------------------- more information about FDB
```

```
Contents of filename block in FDB

Fid:        0      File ID
            0
            0
Fnam: NEWDIS       File name
Ftyp:     PAS      File type
Fver:       0      Version number

                   :
                   :
                   :————————— more information about file name block

Dvnm:      DR      Device name
Unit:      16.     Unit number

Error occurred at line 21 in program compare
```

Error Termination Status

Both the Pascal-2 compiler and Pascal programs return a termination status when they exit. The Pascal-2 compiler terminates with a "severe error" status if it detects compilation errors. Upon detecting an error while running, such as "subscript out of bounds," a Pascal program also terminates with a "severe error" status. Otherwise, a "successful completion" status is returned.

The termination status can be used by command files or command procedures on RSX-11M systems and on VAX/VMS. For instance, a command file that compiles and task-builds a Pascal program can use the compiler termination status to detect any errors and skip the task-build step. In the same way, command files or the batch processor can detect Pascal programs that abort.

The program termination status is also returned to tasks that spawn other tasks. In this case, up to 16 bits of information may be returned.

The Pascal support library contains a routine that may be called from Pascal programs to set the termination status and stop the program. To use this feature, declare an external procedure named `exitst`. This procedure, defined in the support library, takes an integer argument, as shown:

```
procedure Exitst(Status: integer);
 { procedure declaration }
   external;
```

Call the procedure at a point in the program where you want to exit in case of a severe error, as shown:

```
begin       { program Severe }
  :
  Exitst(4);------------- terminate with severe status
  :
end.        { program Severe }
```

A status of 1 means normal termination; any other status means that an error terminated the program.

Implementation Notes

The features described in this section are extensions of standard Pascal to be used solely with Pascal-2.

Multiple Source Files

To combine multiple Pascal-2 files into a single compilation unit, you may use multiple input files on the compilation command line, the `%include` extended language feature within the program text, or both.

The choice depends on the need. If, for instance, you are preparing programs for different machines, you can separate machine-dependent data from your individual programs and use the configuration data in a "header" file on the compilation command line.

The `%include` directive allows the inclusion of separate text files within a program, thus simplifying the calling of external procedures. The directive is written as:

`%include` '*file-name-string*';

The contents of the file specified by *file-name-string* are inserted at the point of the `%include` directive. The string must contain at least the name of the file; if no file extension is specified, .PAS is assumed. In addition to the file name and extension, *file-name-string* can contain the disk volume number, the UIC and the version number of the file.

The single quotes ('...') enclosing *file-name-string* are required if a file version is included as part of the string; otherwise they are optional. In this way Pascal-2 can distinguish the semicolon immediately preceding the version number from the semicolon that terminates the directive. Despite their optional nature, we recommend that you use the single-quote delimiters on all `%include` directives, not just when a version number is included.

```
%include 'hdr';
%include '[33,16]libdef.pas;7';
```

Each included file may itself contain `%include` directives, to a maximum nesting of seven levels.

The example below illustrates the use of both header files and the `%include` directive.

Assume that the source file CONFIG consists of this:

```
{ This file contains configuration data that is }
{ subject to change from installation to installation. }

const
  MaxEntries  = 10;          {entries allowed}
  Debug = false;             {if true, make debugging calls}
```

Assume also that the source file COMDEF consists of this:

```
{ This file contains the definitions of some external }
{ procedures, together with the type declarations needed }
{ by the main program and the external routines. }

const
  NameSize = 24;          {size of name field}

type
  DataItem = record       {describes a customer}
    Name: packed array [1..NameSize] of char;
    Age: 0..maxint
    end;

procedure ReadData(var ThisItem: DataItem; {result of read}
                   var Done: boolean {No more items} );
  external;

procedure WriteData(ThisItem: DataItem {item to write} );
  external;
```

And assume that the source file EXAMPL consists of this:

```
%include 'comdef';



var
  Base: array [1..MaxEntries] of DataItem;
  Buf: DataItem;

  Counter: 0..MaxEntries; {count of items in data base}
  I: 0..MaxEntries;       {induction var}

  Done: boolean;          {set when no more items}


begin
  Counter := 0;
  repeat
    ReadData(Buf, Done);
    if not Done then begin
      Counter := Counter + 1;
      Base[Counter] := Buf;
      end;
  until Done;

  { Process data base }

  for I := 1 to Counter do
    WriteData(Base[I]);
end.
```

These files are compiled with the command:

```
>PAS CONFIG,EXAMPL
```

The result is an object module, EXAMPL.OBJ, containing the output from the compilation of CONFIG, COMDEF, and EXAMPL, concatenated in that order. The object module can then be processed through the Task Builder to produce an executable image.

Any compilation switches apply to all input files.

Access to Files and Records

Pascal-2 only supports the Files-11 software system for accessing files. Files must be specially formatted volumes associated with disks, DECtapes, or magnetic tapes. For details, see the *MCR Operations Manual.* Files-11 is incompatible with Record Management Services (RMS).

Although Pascal-2 does not support RMS, an optional RMS interface allows you to access files and records from within a Pascal program by calls to standard procedures. The *User's Guide for RMS-11 Pascal-2 Interface* provides details of the software's use.

Local Files Closed on Procedure Exit

Consider a procedure (or function) that opens one or more files local to that procedure. Assume that the file variable for the file is defined local to that procedure. When the procedure exits and returns to the calling routine, all files defined local to that procedure are closed. This convention is necessary because upon procedure exit all local variables are deallocated. Once local variables are deallocated, they cannot be referenced again. Therefore, if a local file variable is deallocated, your program can no longer access that file, and no other program may access the file until your program terminates.

To prevent files opened in a procedure from being closed upon procedure exit, define the file variable as a global variable. Since the file variable is in the global data area, the file remains open and accessible until the file is explicitly closed or the program terminates.

Specifying the Location of The Compiler's Work Files

The Pascal-2 compiler opens several temporary scratch files when it compiles a program. For large Pascal programs these files can become quite large (several hundred blocks) and they can be used quite heavily. The V2.1 compiler attempts to open its scratch files on the logical device called `WK:`. If this logical device does not exist, the scratch files are opened on `SY:`, the system device.

If you are running on a multi-disk system, and the disk you are using has very little free space, you can assign WK: to some other disk that has more room. Use the ASN command to do this, as shown below:

```
>ASN DL1:=WK:
```

This command associates the disk DL1: with the logical name WK:. The compiler then opens its scratch files on DL1:.

If your system has different kinds of disks, you should assign WK: to the fastest disk on your system. This reduces compilation time for large programs. You can experiment by using the times compilation switch to see if there is a significant change in compilation times with various disks on your system.

The Use of 'SY:'

On RSX systems, the logical device SY: defines the device on which files are opened when no explicit device is given in the file specification. When you log on to your system, SY: is assigned to your default device. You can modify the device associated with SY: by using the ASN (assign) command.

When a Pascal task is initialized, the Pascal support library determines the physical device associated with SY:. Files opened by the program are opened on this device by default. This assures that files are opened on the correct device even if SY: is assigned to a different device while the Pascal task is executing.

Variable Initialization

The Pascal standard states that variables must be initialized before they are used. Otherwise, their values are unpredictable. Pascal-2 catches most uninitialized variables but can't possibly flag all of them. In short, variable initialization is the programmer's responsibility.

Obvious cases are easily detected, but more complex violations such as the initialization of I below are not caught by the compiler.

```
var
  I, K: integer;

begin
  read(K);
  if K <= 0
    then I := 0———————— variable is initialized here
  else I := I + 1;——————————— but not here
end.
```

Reading MCR Command Lines

To process MCR command lines, use the Pascal support library procedure GMCR. This procedure reads the command line and makes it available to the program as a line of input. If GMCR is not called, the command line is ignored and input comes from the terminal.

GMCR is declared as an external with no parameters, as shown:

```
procedure GMCR;  external;
```

If you wish to process the command line, call GMCR at the beginning of your program, before any input is read. This call obtains the MCR command line for your use.

After GMCR is called, check the value of the predefined file variable `input` to determine whether a command line is present when your Pascal program is executed. If `input^` (the next available input character) is a blank when your Pascal program starts, then no command line is present. If `input^` is not a blank, then the MCR command line is present as the first line of input.

The GMCR routine cannot be placed in a resident overlay or a shared resident library, and it should not be placed in a disk overlay area. However, it is possible to use the routine by loading it into the root of a program. For the root segment of a program called TEST, you can do this with the following TKB command fragment:

```
...test-lb:[1,1]paslib/lb:$gtmcr...
```

The GMCR routine works fine in the program root and doesn't require much room.

The following sample program, FPRINT.PAS, shows a way to check for and read a file name specified on an MCR command line. Note that the name of the program is also part of the command line. After it has read the file name, the program simply prints the contents of the file.

Example: Program To Print a File's Contents (FILEPRINT.PAS)

```
program FilePrint;

  const
    FilenameLength=80;          { max number of chars in a name}

  var
    Filename: packed array [1..FilenameLength] of char;
    Ch: char;
    I: integer;

  procedure GHCR; external;          { procedure to get command line }

  begin
    GHCR;                            { make command line available }
    if input^ <> ' ' then begin      { command line present }
      repeat                         { skip over command }
        read(Ch);
      until (input^ = ' ') or eoln;
      if not eoln then
        repeat                       { ignore blanks after command }
          read(Ch);
        until (input^ <> ' ') or eoln;
      end;
    if (input^ = ' ') or eoln
      then write ('File to print? ');
    if eoln then readln;             { command line but no file name }
    readln(Filename);                { filename from user or cmd line }
    reset(input,Filename);           { open the input file }
    while not eof do begin           { once per line }
      while not eoln do begin        { once per char }
        read(Ch); write(Ch);
        end;
      readln; writeln;
      end;
end.
```

Compile and task-build the above program with these steps:

```
>PAS FPRINT
>TKB
TKB>FPRINT/CP/FP=FPRINT,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>TASK=...PNT
TKB>//
>INS PNT
```

The TASK option gives a unique name to the task so that, when installed via the INS command, the task can be activated with the command PNT.

Once installed, the PNT task can be invoked in three different ways. If the RUN command is used to execute the task, no command line is present, so the program will ask the user which file is to be printed. Or, you can activate the task by typing the name of the task, PNT. The program checks for the case in which the task is invoked with a command line, but no file name is given. As in the previous case, the user is prompted for the name of the file to print. Finally, the program can be initiated with a command line that also contains the name of the file to print. The examples below show the different ways to run PNT.

```
>RUN FPRINT
File to print? X.LST
          :
          :
     { contents of X.LST }
          :
          :
>PNT
File to print? X.LST
          :
          :
     { contents of X.LST }
          :
          :
>PNT X.LST
          :
          :
     { contents of X.LST }
          :
          :
```

Executing MCR Commands from Pascal Programs

Using the procedure DoCmd (Do Command) provided below, users can execute MCR and other system commands from within Pascal programs. The procedure provides an easy way to execute any one-line command as if you had typed the command at a terminal.

To give you an idea of its capabilities, DoCmd can perform tasks such as:

- Run PIP to print, rename, delete, copy, or purge files.
- Run PIP to get a directory listing of some UIC and write it to a file. Then open that file in a Pascal program and perform some operation on each file.
- Have the Pascal program create and close a command file (say, TEST.CMD), which can then be passed to the indirect command file processor with the command @TEST.

- Send a series of commands to allocate a floppy drive, search for bad blocks, initialize, and mount a floppy.
- Run the Pascal compiler, and, if the compilation contains no errors (status word=1), run the Task Builder.
- Change the system date, create or delete partitions, set terminal characteristics.

To use **DoCmd**, you must first create the file DOCMD.PAS (next page), then compile it. Note that the source file contains the **$nomain** embedded switch. This defines **DoCmd** as an external Pascal procedure. **DoCmd** performs three operations: detaches the user's terminal; spawns the MCR, passing it a command line; and then returns the execution status of the completed task. Each of these operations is explained in the following paragraphs.

DoCmd calls the **detach** procedure to detach the user's terminal so that the spawned task can access the terminal. **Detach**, defined in the Pascal support library, is explained in "Detaching From the Terminal" later in this guide. The logical name **TI:** in the spawned task will be the terminal (**TI:**) of the task that called **DoCmd**.

After detaching from the terminal, the **DoCmd** procedure initializes the variables required for the **SPAWN** system call. **DoCmd** allocates a 8-word status return block, **Sts**, of type **StatusBlock**. The first field of the return block, **ExitStatus**, is the termination status of the spawned task where '1' means a successful completion and '4' means a severe error aborted the spawned task. The second field, **TKTN_Code**, is the system abort status that is returned if RSX aborts the task before it can return a status. The remaining seven words of the block are unused and reserved for future use. **SPAWN**, an RSX system directive, initiates the MCR and executes the desired command. Null parameters are used in place of optional parameters on the call. See "Calling FORTRAN Subroutines from Pascal Programs" for information on null parameters.

The **WAITFR** system directive, also a FORTRAN call, waits for event flag number 1 to be raised, signalling the completion of the spawned task.

Upon return, the **SPAWN** directive sets the status words.

The code for DOCMD.PAS is:

Example: Executing System Commands From Within Pascal Programs (DOCMD.PAS)

```
program DoCmd;
 {$nomain}————————————— signifies an external procedure

  type
    StatusBlock = record       { status of offspring }
        ExitStatus: integer;   { task exit status }
        TKTN_Code: integer;   { TKTN abort code }
        Unused: array [2..7] of integer;
      end;

    CommandLine = packed array [1..79] of char; { a command line }
    TaskName = array [1..2] of integer;  { task name }

  var
    Null origin 177777B: integer;  { null parameter for Fortran }

  procedure SPAWN(var Task: TaskName; { task name }
                  var Group: integer; { group UIC code }
                  var Member: integer; { member UIC code }
                  var Efn: integer; { EFN to set when offspring terminates }
                  var AST: integer; { AST address }
                  var Status: StatusBlock; { offspring status }
                  var Parm: integer; { address of status block if AST }
                  var Cmd: CommandLine; { command to execute }
                  var CmdLen: integer; { length of command line }
                  var Unit: integer; { terminal unit number }
                  var Dnam: integer; { device name mnemonic }
                  var DirectiveStatus: integer { Directive status } );
    nonpascal;  { system routine to spawn a task and pass it a command line  }

  procedure WAITFR(var Efn: integer);
    nonpascal;  { system routine to wait for EFN to be set }

  procedure Detach;
    external;  { detach from the terminal }


  procedure DoCmd(var Cmd: CommandLine;
                  CmdLen: integer;
                  var Status: integer);
    external;  { procedure to execute a command line }
```

```
procedure DoCmd; ———————————— required for external modules

  var
    Task: TaskName;
    Efn, DSW_Status: integer;
    Sts: StatusBlock;

  begin
    Detach;                    { detach from the terminal }
    Task[1] := 50712B;         { Rad50 for "MCR" }
    Task[2] := 131574B;        { Rad50 for "..." }
    Efn := 1;       { use event flag }
    Sts.TKTN_Code := 0;
    Sts.ExitStatus := 0;
    SPAWN(Task, Null, Null, Efn, Null, Sts, Null, Cmd, CmdLen, Null, Null,
          DSW_Status);
    if DSW_Status <> 1 then Status := DSW_Status  { error }
      else begin
        WAITFR(Efn);           { wait for EFN to be raised }
        if Sts.TKTN_Code <> 0 then Status := Sts.TKTN_Code
          else Status := Sts.ExitStatus;
        end;
    if Status = 0 then Status := 1;
  end;
```

DoCmd requires three parameters:

DoCmd(*Cmd*, *CmdLen*, *Status*);

where

Cmd is a packed array of characters containing the command line to send to the MCR for execution. The maximum length of the array is 79 characters.

CmdLen is the length of *Cmd*, in bytes. This integer must lie in the range 1..79.

Status is an integer variable for the returned status of *Cmd*'s execution. Any error-numbering convention can be defined for each program. The default convention is as follows:

Status	Indication
1	Normal completion of task
2	Non-fatal error, task ran to completion
4	Severe error, task aborted

For users setting their own conventions, the procedure **exitst**, defined in the Pascal support library, accepts a single integer parameter. This procedure is used to terminate a Pascal task and return a 16-bit status value. If the **SPAWN** directive is rejected, the status word contain the directive status word (DSW) code.

Compile DOCMD.PAS with the following command. (The nomain compilation switch is embedded in the code.)

```
>PAS DOCMD
```

After it is compiled, the DoCmd procedure can be linked with any Pascal program.

The sample program, CMDTST.PAS, below, shows you how RSX command lines might be issued from within a Pascal task. The program consists of the external procedure DoCmd and the main program. Only one command is executed in this example for simplicity. When the command is completed, the status word is printed.

Example: Issuing Command Lines From a Pascal Program (CMDTST.PAS)

```
program CmdTst;
  type
    CommandLine = packed array [1..79] of char;

  var
    Cmd: CommandLine;
    Cmdlen, Status: integer;

  procedure DoCmd(var Cmd: CommandLine;
                  Cmdlen: integer;
                  var Status: integer);
    external;

  begin     { CmdTst }
    write('Command? ');
    readln(Cmd);
    Cmdlen := 79;
    while (Cmd[Cmdlen] = ' ') and (Cmdlen > 1) do
      Cmdlen := Cmdlen - 1;
    DoCmd(Cmd, Cmdlen, Status);     { execute the command }
    writeln('status=', Status: 1, '.');
  end.      { CmdTst }
```

Compile and execute the program with these commands. The spawned task is the system TIME facility.

```
>PAS CMDTST
>TKB CMDTST/FP/CP=CMDTST,DOCMD,LB:[1,1]PASLIB/LB
>RUN CMDTST
Command? TIME
14:57:02 29-JUN-1983
status=1.
```

NOTE

On some RSX systems, the Task Builder may detect an undefined symbol, `$EXIT`, the FORTRAN library routine that terminates a FORTRAN program. This error has no bearing on Pascal programs and can be ignored.

Lazy I/O

Pascal-2 uses an input interface known as "lazy I/O" to handle input from `text` files. In order for a program to receive information from an input file, including from an interactive file system such as your terminal, the program must be able to determine the current status of the file. More specifically, it must be able to retrieve current values of `eoln` and `eof` and the current record from the file's buffer variable (`F^`). A program must therefore wait for a full line to be entered before it can deteremine "end of line" (or "end of buffer") and be able to interpret the results. The function of lazy I/O is to safely delay any input operation until the program can process a full line.

When a Pascal program requests an input operation on a text file, the operation is recorded for later use. The delayed operation is triggered by any subsequent reference to the file's buffer variable, the `eof` or the `eoln` value. The delay is invisible to the program but you see it by the way the program is synchronized with interactive input.

You need to be aware of the effect that lazy I/O has on synchronization of input and output operations. As an example, consider a simple program that reads its standard file `input`, which is connected to a terminal. The program prompts for each line and stops at the end of the file. The design of the program is dictated by two requirements:

1. For the prompt to be effective, it must appear before the user is required to type the line.
2. To detect the end of the file correctly, the program must check for it before reading each line.

To meet both of these requirements, the program must print the prompt before performing any operation that requires the next line of the file to be known: checking for "end of file" or reading the line. The following example shows a typical implementaion of lazy I/O. A Control-Z (`^Z`)at the prompt signals an "end of file" on the interactive input file, halting the program.

Example: Use of Lazy I/O

```
program Interactive;

begin
  write('prompt:');  ———————— prompt appears before an input line is required
  while not eof do   ———————————— eof check occurs before the line is read
  begin
    readln;
    :
    :
    :———————————————————————————— process the input line
    write('prompt:');
  end
end.
```

Terminal I/O

The RSX terminal driver acts as an intermediary between a Pascal program and the terminal that is running that program. The terminal driver is record-oriented, which means it sends a full line of text to the screen or to the program instead of one character at a time.

For example, when a **read** statement reads input from a terminal, the terminal driver reads each character and places it in an internal buffer until the terminal driver encounters an end of line (a carriage return). At this point, the terminal driver sends the entire line to the program and lets the program process the line one character at a time.

The same is true for **write** statements, except that the support library buffers the characters being sent to the terminal until a **writeln** sends the buffer to the terminal driver or until the buffer fills. Then the record is displayed on the screen. The buffer size can be controlled with the **/buff:*n*** (buffer size) file control switch. See "Single-Character I/O," below.

As mentioned above, the **writeln** statement instructs the terminal driver to send the output to the screen. After the terminal driver prints a line, it positions the cursor over the first character of the line it just printed. The usual output sequence sent by the RSX terminal driver is: line feed, data, carriage return. This sequence moves the cursor to the next line, prints the data and returns the cursor to the first position of the newly printed line. This method of writing lines to a terminal is different from other operating systems where the normal output sequence is data, carriage return, and line feed (print the record, move to the first character of the line, then move to the next line).

The normal sequence of commands issued by the terminal driver may not particularly suit special I/O applications such as direct cursor addressing. Sometimes the line-feed and carriage-return characters are often not needed or are unwanted. To gain control of the terminal-I/O command sequence, use the **/fta** file control switch (see below). This feature allows you to override the effects of the terminal driver.

FORTRAN Carriage Control

Pascal-2 allows your Pascal programs to write **text** files that follow the FORTRAN standard output conventions. To do this, specify the **/ftn** file control switch on the **reset** or **rewrite** statement that opens the file. For interactive I/O, the **/ftn** switch is used with the standard file **output**.

FORTRAN conventions state that the first character of each line of an ASCII file is nonprintable and is used to control the vertical formatting of an output file (or terminal screen). Most characters have no useful meaning in the first position, but some characters significantly affect the format of the output.

The table below summarizes the most commonly-used vertical formatting characters. A complete list of these characters is in the *RSX-11M/M-Plus I/O Drivers Reference Manual.*

Character	Meaning	Output Sequence
<space>	Normal output	Line feed, data, carriage return
0	Double-space	Two line feeds, data, carriage return
1	Page eject	Form feed, data, carriage return
+	Overprint	Data, carriage return
$	Prompt	Line feed, data, remain on same line
<null>	No special formatting	Data only

The null character (**chr(0)**) can be used to prevent the terminal driver from adding any special characters to the data you write to the terminal. You have to insert the carriage-control characters yourself, as shown in the last line of program FTNOUT.PAS, listed below. (**Chr(13)** is the carriage-return character, and **chr(10)** is the line-feed character.)

This example program shows how to use the vertical formatting characters from a Pascal program. The first character written on each line is interpreted as the vertical-format control character.

Example: Program Using FORTRAN Control Characters

```
program FtnOut;

 begin
   rewrite(output, 'TI:/ftn');
   writeln(' Normal output');
   writeln(' More normal output');
   writeln('0Double space');
   writeln('1Page eject');
   writeln(' Normal output of a long line');
   writeln('+Overprint*****');
   writeln('$Prompt output: ');
   writeln(chr(0), 'Internal format', chr(13), chr(10));
 end.
```

Running this program produces this output on your terminal:

```
>RUN FTNOUT
Normal output
More normal output

Double space









Page eject
Overprint**** of a long line
Prompt output: Internal format

>
```

In the above listing, the "overprint" `writeln` replaces the characters "Normal output" (from the previous `writeln`) with "Overprint****," resulting in a different line of text. If you set your terminal to a slow baud rate, you would actually see the overprinting occur. If you direct the file to a printer, the overprinted line will contain overstrikes.

The `/ftn` switch does not solve all terminal I/O problems. The data is not written to the terminal immediately; the characters are still buffered until a `writeln` is executed. You can overcome this problem by using the feature that allows single-character I/O (see following).

Single-Character I/O

The `/buff:`*n* I/O control switch sets the maximum size of each line of a `text` file to *n* bytes. By setting your terminal's internal line length to 1 with `/buff:1`, you can write programs that perform single-character input and output. This feature is useful when you need to do special output formatting to a video terminal, such as direct cursor addressing of output on the screen.

To enter single-character mode, specify the `/buff:1` switch on the `reset` statement for single-character input and on the `rewrite` statement for single-character output. In single-character mode, the terminal driver reads and writes the characters as usual. But since the internal buffer size is one byte, each new character fills the buffer, causing the buffer to be emptied. In other words, each character written via `write` is immediately printed on the screen; each character that you enter is immediately sent to the program.

No special formatting characters are inserted when you use the `write` statement (see "FORTRAN Carriage Control," above). A `writeln` statement prints a carriage return followed by a line feed, as if the buffer were empty. You can supply formatting characters by using the **chr** function to generate the appropriate characters. For single-character input, you do not need to type a carriage return after each character to signify the end of the line.

The following sample program combines both single-character input and single-character output. The program simply echoes each character as it is entered.

Example: Use of Single-Character I/O

```
program Single;
  var
    Ch: char;

  begin
    reset(input, 'TI:/buff:1');
    rewrite(output, 'TI:/buff:1');
    write('Type a message: ');
    while not eoln do
      begin
        read(Ch);
        write(Ch);
      end;
  end.
```

After compiling and linking this program, run it to get these results:

```
>RUN SINGLE
Type a message: ddoouubbllee  vviissiioonn
```

Two other file control switches, when used with **/buff:1**, enhance the use of single-character I/O. The **/ral/buff:1** switch combination enables single-character binary input for the reading of control and escape characters such as Control-C. The **/noecho/buff:1** switches temporarily disable the terminal's character echoing capability. See "I/O Control Switches" for details.

Since single-character input/output vastly increases system overhead, this feature should be used with care. Overhead is increased because the monitor is called for each character read from and written to the terminal. Several users simultaneously outputting in single-character mode can significantly reduce the system response time.

Detaching From The Terminal

Pascal-2 programs attach the user's terminal (TI:) on Logical Unit Number 5 to prevent input from going to the MCR or elsewhere on heavily loaded systems. This also enables the Control-O (^O) feature to prevent the printing of unwanted output.

While attaching the user's terminal is an advantage for interactive Pascal programs, it can be a problem for non-interactive programs because Pascal-2 programs reserve the terminal for the exclusive use of the task. This prevents the user from starting multiple tasks from the same terminal. To free the terminal for other uses, you can have the Pascal program detach from the terminal via an external procedure defined in the Pascal support library. The sample program below shows how to call this procedure.

```
program Det;
procedure Detach; external;  { defined in library }
begin                        { start of main program }
  Detach;                    { make terminal available }
    :
 { rest of program goes here }
    :
end.
```

This program calls the support library routine **detach** to detach from the terminal as soon as the program starts. The detaching should have no effect on the performance of the Pascal program. If the program is very interactive, however, you may get an error message or an MCR prompt if you respond too quickly to a Pascal request for input.

See "Executing Commands from Pascal Programs" for a use of **detach**.

It is possible to "patch" the symbol **p$natt**, a global variable in the support library, at task-build time. With this patch, a program never attaches to the terminal. When you use this patch, you do not need to use **detach**. See the "Support Library" section for details.

Pascal-2's Use Of LUNs

A logical unit number, or LUN, is a number the Pascal support library and RSX use to associate a file variable with the physical device storing the file. The Task Builder allocates six LUNs to a task, by default. Of those six, the Pascal support library assigns two LUNs for the standard files **input** and **output**. Therefore a typical program can open up to four files without aborting with the "too many files open" run-time error.

Normally, you do not need to know how LUNs are assigned in your program; all file accessing is handled automatically by the Pascal support library. However, in some special applications, knowledge of LUN assignments may be important. For instance, if you wish to open a file using Pascal and then access the file with specialized MACRO subroutines, you may need to control the LUN assignment for the file. The same is true if a Pascal program calls FORTRAN subroutines that perform FORTRAN I/O. In this case you may find it necessary to control which LUNs are being used by your Pascal program, to prevent FORTRAN LUN assigments from interferring with Pascal-2's LUN assignments.

To avoid conflicting LUN assignments, specify the `/lun:n` I/O control switch on the `reset` or `rewrite` statement that opens the file. If LUN n is already in use, the error message "LUN already in use" results. For example, the following statement opens a file on LUN 3:

```
reset(F, 'sample.dat/lun:3');
```

The `/lun:n` switch is discussed in the "I/O Control Switches" section earlier in this guide.

When the program is initialized, the support library assigns LUN 5 for interactive input operations and LUN 6 for interactive output operations to the logical device `TI:` (your terminal). In addition, the terminal is attached on LUN 5 to reserve the terminal for the exclusive use of the Pascal program. (See the previous subsection for instructions on how to detach from the terminal from within a Pascal program.)

As each file is opened with a `reset` or `rewrite`, the support library assigns the next available logical unit number to the file, starting with LUN 1. The file is accessed via that LUN until the file is closed, at which time the LUN is available for reassignment. Keeping in mind that `input` and `output` are LUNs 5 and 6, the first file a typical program opens is assigned to LUN 1, the second file to LUN 2, and so on until the fifth file, which is assigned to LUN 7. If the default allocation is used, the fifth file cannot be opened because there are no more LUNs available for the task to use.

To increase the number of logical units available to a task, use the `UNITS` Task Builder option. This is necessary if your program opens five or more files or if you are linking the Debugger or Profiler with your program. (The Debugger and the Profiler open five files each. See the Debugger section for details.) For the `UNITS` option, 20 is a reasonable value:

```
>TKB
TKB>LUNTST/FP/CP=LUNTST,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=20
TKB>//
```

Extra units do not increase the size of the task significantly. The maximum number of logical units available to a task is 250.

For more information on the support library and a way to "patch" the standard files `input` and `output` to use different LUNs, see the "Support Library" section in this guide.

Using Event Flags

Tasks use event flags to learn whether a specific I/O operation, or "event," has completed. Tasks may set, clear and test event flags with special system directives. These directives are explained in the *RSX-11M/11M-PLUS Executive Reference Manual.* Users who write programs that call system services or perform special I/O operations may need to know how Pascal–2 uses event flags.

Event flags are commonly used to signal the completion of I/O operations and as a timer to signal the passage of time. Event flags may be shared among several executing tasks.

Of the 96 event flags available on the system, user programs may set, clear or test event flag numbers (EFNs) 1 through 24 for flags local to a task, 33 through 56 for flags common to all tasks, and 65 though 96 for flags known to a specific "group" of tasks. The remaining EFNs (25 through 32 and 57 through 64) are reserved for the system. All event flag numbers are expressed as decimal integers.

The Pascal support library uses EFN 17 to monitor completion of Pascal I/O operations. Therefore, user programs cannot use event flag number 17 unless the support library's EFN is changed. See "Changing the Support Library's Event Flag" earlier in this guide for information on how to change the EFN used by the support library.

To use an event flag, define an integer variable to contain the event flag number. This variable is used throughout the program in operations involving the event flag. The example below shows the use of event flag numbers. Program `Delay` sets a timer to raise EFN 1 after a specified interval of time. To set the timer, the program calls the system service routines `MARK`, which actually initiates the timer and associates the event with EFN 1, and `WAITFR`, which waits a specified number of seconds for the event flag to be set before returning to the main program. While waiting, the program could execute code that is not dependent upon completion of the timed operation.

Example: Use of Event Flags

```
program Delay;

  type
    TimeInterval = (Unused, Ticks, Seconds, Minutes, Hours);

  var
    Interval: TimeInterval;        { Time interval unit }
    WaitTime: integer;             { How long to wait }
    Efn: integer;                  { Event flag to set }

  procedure MARK(var Efn: integer;        { Event flag number }
                 var DelayTime: integer; { How long to wait }
                 var Unit: TimeInterval  { Time units } );
    nonpascal;   { system procedure to set a timer }

  procedure WAITFR(var Efn: integer);
    nonpascal;   { system procedure to wait for an event flag }


  begin   { Delay }
    write('How many seconds to wait? ');
    readln(WaitTime);
    Interval := Seconds;
    Efn := 1;
    MARK(Efn,WaitTime,Interval);
    writeln('Waiting...');
    WAITFR(Efn);
    writeln('Done waiting');
  end.   { Delay }
```

Random Access to 'Text' Files

The **seek** procedure cannot compute the location of a particular record (line) within a file of type **text** because the lines are of variable lengths. The Pascal support library supplies two external procedures, **getpos** and **setpos**, that simulate random access to **text** files.

These two procedures are not predefined and must be declared in your program as **external**. **Getpos** determines the starting location of the next line of a file, and **setpos** sets the file pointer to the specified starting location of a line within the file. The beginning of each line of a text file is denoted by a block number and a byte offset into that block. Each block contains 512 bytes. The first line of a file starts with block 1, offset 0. If you try to access a nonexistent position or a position in the middle of a line, an I/O error will result.

The block number and byte offset must be values returned by **getpos**. You cannot compute the values yourself. When the file is being read, use **getpos** to determine the starting position of the next line and save that block and offset combination for later use by **setpos**.

Bear in mind that this is not "true" random access; you cannot access individual characters, only individual lines of text. Use your Pascal program to access characters individually within each line.

Procedure 'GetPos'

Procedure **getpos** determines the starting position of the next line to be read from or written to a text file. **Getpos** requires three parameters, passed by reference, as shown below:

```
procedure GetPos(var F: text; var Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the returned disk block number of the next line in file *F* to be read or written.
Offset	is the returned byte offset into *Block*. Together, *Block* and *Offset* point to the next line to be processed.

You should always call **getpos** to obtain the location in the file before you call **setpos**, so the block and offset values being passed to **setpos** are valid.

The example in the next subsection shows the use of **getpos**.

Procedure 'SetPos'

Procedure **setpos** positions the file pointer to a specified block number and byte offset into that block. **Setpos** accepts the same three parameters as **getpos**, except *Block* and *Offset* are passed by value. The **setpos** declaration is as follows:

```
procedure SetPos(var F: text; Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the block number to which the file pointer is set.
Offset	is the byte offset into *Block*. Together, *Block* and *Offset* point to the new position.

To stress an earlier point, the block number and byte offset must be values returned by **getpos**. Do not attempt to compute the values yourself. Save the returned values for later use.

If an error is detected while **setpos** tries to position the file, the end-of-file flag **eof** is set to true. The **ioerror** and **iostatus** support library functions may help you to determine the reason that the line could not be accessed. (For details on **ioerror** and **iostatus**, see "I/O Error Trapping" in this section.) If a file is positioned to a block and offset that does not correspond to the first character of a line, the results are unpredictable.

The example below shows a way to use getpos and setpos. Program Reverse reads a text file and saves the position of each line in a linked list. It then prints the file in reverse line order so that the last line of the file is printed first and the first line is printed last.

Example: Use of 'GetPos' and 'SetPos' Procedures

```
Program Reverse;

  type
    Pointer = ^position;
    position =
      record
        Next: pointer;
        Block: integer;
        Offset: integer;
      end;

  var
    F: text;
    Filename: packed array [1..80] of char;
    P, X: pointer;
    Done: boolean;
  procedure GetPos(var F: text;
                   var Block, Offset: integer);
    external;

  procedure SetPos(var F: text;
                   Block, Offset: integer);
    external;
```

```
begin
   write('File name? ');
   readln(Filename);
   reset(F, Filename);
   P := nil;
   repeat ---------------------------------- read the file
     new(X);
     with X^ do GetPos(F, Block, Offset);----------- get start of next line
     X^.Next := P;
     P := X;
     Done := eof(F);
     if not Done then
       readln(F);
   until Done;
   while P <> nil do ------------------------ write the file
     with P^ do
       begin
       SetPos(F, Block, Offset);
       if not eof(F) then
         begin
         while not eoln(F) do
           begin
           write(F^);
           get(F);
           end;
         writeln;
         end;
       P := P^.Next;
       end;
end.
```

Unsigned Integer Conversion

On the PDP-11, integer variables are stored in 16-bit words. These 16-bit words may be interpreted as signed or unsigned integers. A signed number, in two's complement notation, represents numbers in the range −32767..32767. An "unsigned" (also called "extended-range") number by definition does not have a sign bit; rather, it uses all 16 bits to represent an integer in the range 0..65535.

When their values are compared or used in mathematical expressions, unsigned integers differ greatly from signed integers. As an example, consider a word in which all 16 bits are set to one. This word has a value of -1 when interpreted as a signed integer, or a value of 65535 when interpreted as an unsigned integer. When this word is compared with some other value, the PDP-11 uses different combinations of instructions for signed and unsigned comparisons. If this number is multiplied by two, the result is a value of -2 for signed or 131070 for unsigned. The latter is an overflow condition because the result does not fit within 16 bits.

The Pascal-2 compiler and support library also differ in their treatment of signed and unsigned integers. When you define a variable to be of type integer in your Pascal program, the compiler treats that value as a signed integer, unless you specify an unsigned integer using a subrange notation such as:

```
type
  unsigned = 0..65535;

var
  I: unsigned;
```

According to your data declarations, the compiler generates the correct code to compare, multiply, or divide unsigned numbers. The compiler can then deal with unsigned integers.

The Pascal support library uses a single routine to print integers. If you attempt to write out the value of an unsigned integer, you find that the number is always treated as a signed integer. This routine interprets all integers as signed values. If you want to write out the value of an unsigned integer, use the following procedure in your program instead of the write statement. This procedure, Uwrite, takes an unsigned integer and a field width as arguments. The number is printed as a value in the range 0..65535, right justified in the specified field.

Example: Procedure for Writing Unsigned Integers

```
procedure Uwrite(I: unsigned;
                 Width: integer);

  { This procedure writes an unsigned integer to output. }

  begin  { Uwrite }
    if (I > 32767) and (Width >= 0) then
      begin
      if Width > 0 then
        Width := Width - 1;
      write(I div 10: Width);
      I := I mod 10;
      Width := 1;
      end;
    write(I: Width);
  end;  { Uwrite }
```

The PDP-11 floating-point hardware uses signed conversion when it converts an integer value to a real value. If you wish to convert an unsigned integer to real, use the following function. This function, Ufloat, takes an unsigned integer as its argument and returns a real value in the range of 0.0 to 65535.0.

Example: Procedure To Convert Unsigned Integer To Real

```
function Ufloat(X: unsigned): real;

  { This function converts an unsigned number to a real number. }

  var
    R: real;

  begin  { Ufloat }
    R := X;
    if R < 0.0
      then R := R + 65536.0;
    Ufloat := R;
  end;  { Ufloat }
```

The trunc and round functions convert real numbers to integers. Since the floating-point hardware assumes a signed conversion, the following function should be used when an unsigned integer result is desired. The function Utrunc takes a real number in the range 0.0 to 65535.0 and converts it to an unsigned integer.

Example: Procedure To Convert Real To Unsigned Integer

```
function Utrunc(R: real): unsigned;

  { This function converts a real number to an unsigned integer. }

  begin  { Utrunc }
    if (R > 65535.0) or (R < 0.0)
      then writeln('Unsigned number out of range');
    if R > 32767.0
      then R := R - 65536.0;
    Utrunc := trunc(R);
  end;  { Utrunc }
```

The unsigned round function is very similar to the above unsigned trunc function.

Multiuser Tasks

The multiuser-task feature is available only on RSX-11M-PLUS systems and on VAX/VMS systems in compatability mode.

A multiuser task is one in which read-only instruction and data program sections (psects) are shared among two or more users, saving a considerable amount of memory if several users run the same task at the same time. With multiuser tasks, each copy of the task has its own read/write psects but only the first copy executed contains the shared read-only code. All other copies of the task reference the read-only code of the first copy.

The information presented below is oriented toward multiuser Pascal tasks. For additional information, read the chapter on multiuser tasks in the *RSX-11M/M-PLUS Task Builder Reference Manual*.

The creation of a multiuser task is very simple. All you have to do is apply the /MU switch to the task image output file, as shown for TEST.PAS. (Only one user is required to do this; others simply run the program.)

```
>TKB TEST/MU/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
```

The Task Builder places the read/write psects in low memory and places the read-only (shared) psects in the highest available multiple of 4K words (an APR). The load map shows this layout. When additional users run the task, the operating system automatically handles the sharing of the read-only code.

One problem you may encounter with multiuser tasks is that less heap storage may be available because of the way the memory management hardware on the PDP-11 allocates the task's read-only memory. As stated earlier, the read-only portion of the task is placed at the high end of memory. However, the amount of memory used for the read-only portion of the task is rounded up to use the next lowest APR, at a loss of heap space. This loss is usually not significant unless a task is very large or requires a large amount of dynamic storage.

For example, if the read-only code and data required 11K words, it is be placed in the task in virtual memory addresses from 20K to 32K. The net result is that 1K of virtual address space is lost. In any case, the savings of physical memory are significant. In the example above, 11K words of memory are saved when each additional user runs the program since they each share one copy of the read-only code.

If your program uses overlays, you should read the chapter on multiuser tasks in the Task Builder reference manual for restrictions concerning multiuser tasks.

Multiple Buffering

Multiple buffering allows Pascal programs to perform I/O operations concurrently with program computations. This overlapping of I/O operations with program computations can significantly improve the performance of Pascal tasks, especially those that handle large volumes of data. A good candidate for multiple buffering is a program that makes heavy use of I/O and performs many computations. Multiple buffering must be enabled on your system before you can use this feature (see below).

Multiple buffering is activated by three file control switches, **/mbf:n**, **/rah** and **/wbh**. (See "I/O Control Switches" for other details on these switches.)

/mbf:n (Multiple Buffering) Specifies the number of buffers to use in accessing the file.

/rah (Read-Ahead) Reads information from the file before it is actually needed by the program, allowing program computations to overlap with file input operations.

/wbh (Write-Behind) Returns control to the program after a write or put statement before the data is actually written to the file, also allowing program computations to overlap with file output operations.

The trade-offs you must consider in determining the optimum number of buffers to use with each file are discussed in "Multiple Buffering for Record I/O" in the *RSX-11M I/O Operations Reference Manual.*

The task's dynamic memory requirements also must be considered. Each file buffer requires about 528 bytes of heap storage. To determine how much heap storage the file buffers will use, multiply the total number of buffers by 528. Adjust this allotment by increasing or decreasing the number of buffers to arrive at the optimum number of buffers for each file.

The following table gives the recommended modes and I/O control switches to use in accessing random access and sequential files having multiple buffers.

File Access Modes

File Type	Operation	Mode	Example
Sequential	`read`	"read-ahead"	`reset(F, 'test.dat/mbf:5/rah');`
Sequential	`write`	"write-behind"	`rewrite(F, 'test.dat/mbf:12/wbh');`
Random access	`read/write`	"write-behind"	`reset(F, 'test.dat/seek/mbf:5/wbh');`

You can use the memmap procedure in the Pascal support library to see the effect of multiple buffering. Memmap displays the contents of the heap, including file buffers. (See "Customizing Error Reporting" in this guide for information on the use of memmap.)

If your program uses mulitple buffering and you link it with an FCS resident library such as FCSRES, you must increase the size of the RSX file storage region to make room for all the file buffers. (FCS allocates file buffers from this region.) Use the EXTSCT Task Builder option to do this. For example:

```
>TKB
TKB>REALNO/FP/CP=REALNO,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>LIBR=FCSRES:RO
TKB>EXTSCT=$$FSR1:nnnnn  ———— size of the file storage region
TKB>//
```

Enabling Multiple Buffering On Your System

Before you can use the multiple buffering feature, you must enable multiple buffering on your system. It is best that you check to see whether multiple buffering is available before attempting to enable it yourself. To enable multiple buffering you must to modify the system I/O library `LB:[1,1]SYSLIB.OLB` by replacing certain I/O modules with the contents of the file FCSMBF.OBJ, in your original RSX sysgen kit. (Consult the *RSX-11M I/O Operations Reference Manual* for more details.)

We recommend that you first make a copy of `LB:[1,1]SYSLIB.OLB`, update the copy to support multiple buffering, test it, and finally replace the new version in `LB:[1,1]`.

```
>PIP SY:/NV=LB:[1,1]SYSLIB.OLB ———————— make copy of SYSLIB.OLB
>LBR SYSLIB/RP=LB:[1,1]FCSMBF ———————— update to support multiple buffers
Module "ANSPAD" replaced
          :
          :
          :———————————————————— other module replacements
Module "WTWATD" replaced
```

The replacement of these modules enables multiple buffering, ANSI magtape support, and support for large buffers (buffers greater than 512 bytes). The commands below will compress the library to remove unused space from SYSLIB.

```
>LBR SYSLIB/CO=SYSLIB
>PIP SYSLIB.OLB/TR
>PIP SYSLIB.OLB/PU
```

It's a good idea to create test programs to be sure that everything is working correctly. In your testing, link your program with the modified copy of SYSLIB.OLB using the `/DL` Task Builder option to change the default system library from `LB:[1,1]SYSLIB.OLB` to the copy of SYSLIB.OLB you modified.

```
>TKB TEST/FP/CP=TEST,SYSLIB.OLB/DL,LB:[1,1]PASLIB/LB
```

After you have verified that multiple buffering works, update the system copy of `LB:[1,1]SYSLIB.OLB`. Once you do this you will be able to use the multiple buffering facility in your programs without any special Task Builder commands.

Resident and Cluster Libraries

A resident library is a collection of commonly used routines from the Pascal–2 support library that can be shared among all Pascal tasks. In the V2.1A release of Pascal–2, a Pascal resident library had to contain modules from both the Pascal support library and File Control Services (FCS). V2.1B and later releases allow users to build resident libraries that do not contain FCS routines. For example, you may want to place both Pascal support library routines, from `LB:[1,1]PASLIB`, and routines for File Control Services (FCS), from `LB:[1,1]SYSLIB.OLB`, in a Pascal resident library.

In many cases, a memory-resident overlaid version of a Pascal resident library can be clustered with a separate FCS resident library. The "cluster library" function permits a Pascal task to use various libraries in the group through one task address window. See the *RSX-11M/M-PLUS Task Builder Manual* for a full explanation of this technique.

A shared resident library has several advantages. First, each Pascal program does not need to have its own copy of the library modules, which reduces memory requirements and improves system performance when several Pascal tasks are executing at the same time. Second, the size of the task image file is smaller because the code for shared routines is not included in each Pascal task image. Also, several compiled Pascal tasks can be stored in less space on your disk. Finally, task-building time is reduced because the modules in the Pascal resident library are pre-linked at the time the library is created. The Task Builder resolves references to library modules directly in the Pascal resident library instead of searching through the Pascal object module library looking for modules to include in your task.

Resident and cluster libraries are created with the Task Builder command file PASRES.CMD that is provided on the Pascal–2 distribution media. You may need to edit PASRES.CMD to tailor the library to meet your needs. See the comments in the PASRES.CMD command file itself and the explanations that follow for guidance.

Creating a Pascal Resident Library

You create a Pascal resident library with the PASRES command file. To use PASRES.CMD, first install your Pascal–2 system using the PASBLD command, making sure that the Pascal support library `LB:[1,1]PASLIB.OLB` is the current version. Then enter the command:

```
>@PASRES
```

The command file asks whether or not you want to build a memory-resident overlaid version of the Pascal resident library.

If your system supports virtual (PLAS) overlays, select this option and the resident library is built as two overlays. The total amount of physical memory required for the library varies between 6K and 8K words, depending upon your hardware configuration. The advantage of virtual overlays is that the 6K and 8K words of library code can be accessed using only 4K words of virtual address space (saving 2K to 4K words.)

If you do not wish to use virtual overlays, PASRES.CMD builds a non-overlaid resident library. This may be necessary on some RSX sytems that do not support such configurations. If virtual overlays are not selected, PASRES.CMD builds a library requiring only 4K words of memory.

After PASRES.CMD builds your library and associated symbol table, you must create a partition and install your resident library file. To do so, you need to know both the task image size of your library file and the base address at which the partition is to be loaded. The task size can be determined by examining PASRES.MAP, which is also created by PASRES.CMD. The value you obtain from PASRES.MAP is given in decimal words. You must convert it to octal bytes, rounded up to the next multiple of 100. For example, a task size of 6464 words converts to 31200 bytes (octal). The base address for the partition is determined by examining the current partition allocation of your operating system, using the PAR utility.

In the following partition allocation sample, the three columns of numbers contain the location of Partition Control Blocks, the base addresses of the partitions in memory, and the length of those partitions, reading from left to right.

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB DRIVER -DL:
       066570 152200 001400 SUB DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

The SET/TOP command is used to reduce the size of the GEN partition by the amount necessary to install the PASRES library. When dealing with partitions, all addresses and sizes are divided by 100 (octal), so if the size needed for your PASRES file is 31200 bytes, you create space for it by entering:

```
>SET/TOP=GEN:-312
```

When you recheck the partition allocation, you see a 31200-byte reduction in the GEN partition.

```
                    :
                    :
GEN    066314 242100 464500 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

The base of the PASRES partition is computed by adding the base of the GEN partition to its length: 242100 + 464500 = 726600. You can now create the PASRES partition based at 7266 with a size 312 by entering >SET/MAIN=PASRES:7266:312:COM to create the following partition structure:

```
                          :
                          :
GEN     006314 242100 464500 MAIN SYS
        042620 242100 020000 SUB  (...MCR)
PASRES  043634 726600 031200 MAIN COM
```

A program designed to use PASRES is compiled in the normal way, but you should use the library option LIBR when task-building to specify that PASRES is used in read-only mode. The following example task builds the program TEST.

```
>TKB
TKB>TEST/FP/CP=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>LIBR=PASRES:RO
TKB>//
```

You use the CLSTR option to specify that PASRES and FCSRES are to be clustered.

```
>TKB
TKB>TEST/FP/CP=TEST,LB:[1,1]PASLIB/LB/$FCSJT,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>CLSTR=PASRES,FCSRES:RO
TKB>//
```

When you use FCSRES, you must allocate space for buffers for all the files that you intend to open. Otherwise you may get an error status of -39 when you try to open a file.

If you are using FCSRES, but not PASRES, space is allocated in the program section $$FSR1 by using the EXTSCT option. In the following example, a buffer of 2100 bytes is created.

```
>TKB
TKB>TEST/FP/CP=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>EXTSCT=$$FSR1:2100
TKB>//
```

When you use PASRES, the ACTFIL option of the Task Builder permits you to specify the number of files you intend to have open at one time.

```
>TKB
TKB>TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>ACTFIL=1
TKB>LIBR=PASRES:RO
TKB>//
```

If the value you specify with `ACTFIL` is greater than one, you must also use the `UNITS` option to make more logical unit numbers available to the task.

PASRES may be similarly clustered with other libraries besides FC-SRES.

Including Pascal Procedures in a Resident Library

The preceding section applies to a general-purpose resident library created from the modules in the Pascal support library. This section you how to include your own compiled Pascal procedures in a resident library so that they can be shared among several Pascal tasks or how to add a new procedure to an existing resident library.

Only Pascal modules compiled as external procedures are allowed in a resident library because their entry points must be available to the Task Builder as global symbols. Main programs are not permitted because they create data areas belonging only to the task that created them, and these areas may not be shared. You can often remove the main program body from a program and compile the remaining procedures as an external module, for placement in a resident library. You can also convert a main program to a procedure, compile it as an external procedure and install it in a resident library.

In addition, all data used by procedures in a resident library must be local data. Where necessary, global variables should be passed as `var` parameters to the external procedures. This restriction applies to external files, including the standard files `input` and `output`. In general, if a file is used explicitly in a procedure in a resident library, the explicit file must be passed as a `var` parameter to the procedure.

Despite a few exceptions, a good rule to follow is: always pass the files `input` and `output` to a procedure in a resident library and always specify the files explicitly on file statements such as `eoln(input)` or `writeln(output)`. Every use of `eoln` and `eof` must include a file argument, so the file `input` must be specified explicitly, as in `eof(input)` and `eoln(input)`. This restriction does not apply to the `read` and `write` (and `readln` and `writeln`) statements. On these statements, the files `input` and `output` can be omitted (implied).

Code placed in a resident library must be compiled with the command line `nowalkback` switch, to eliminate the use of read/write data areas. If a resident library did contain read/write data areas, data in the library could be destroyed by other tasks sharing the data area.

Building a Resident Library

In general, the steps you take in building a resident library are:

1. Create the library source file of shareable code and compile it as an external procedure using the nomain and nowalkback switches.
2. Task-build the object library to produce a loader map. The map file shows you the size of the library task image in words and in octal bytes, two values needed for the PAR (partition) option in Step 4.
3. Use the table provided later in this section to determine the base address of the library, based on the size of the task image.
4. Task-build the object library a second time using the PAR option and the two values derived from the map listing and the table.
5. Create a partition on your system for the resident library. Since only privileged users are permitted to create partitions, we recommend that your system manager do this for you.
6. Install the library task on the system. After this, the library can be linked with any program.

Placing an Existing Procedure in a Resident Library

To convert a procedure in an existing program into a shared procedure, you may placed it place it in a resident library of its own or add it to an existing library.

To trace the steps through an example, let's assume that several users intend to run the simple program WRTSUM.PAS hundreds of times a day. The program calls the Sum procedure to print the sum of 1 and 5. (The answer is 6.) For greater efficiency, you want to create a shared resident library rather than have each user keep a copy of the complete program.

The complete program is:

Example: Use of Shared Procedures

```
program Wrtsum;

  procedure Sum(A,B: integer);
    begin
      writeln('The sum is: ',A+B);
    end;

begin     { main program }
  Sum(1,5);
end.
```

To create the resident library, you must create a separate file to contain the shared code (in this case the procedure Sum). As described above, main programs may not be shared. The file SUM.PAS has these contents:

```
procedure Sum(A,B: integer);
  external;

  procedure Sum;
    begin
      writeln('The sum is: ',A+B);
    end;
```

This module is then compiled as an external procedure.

```
>PAS SUM/NOMAIN/NOWALKBACK
```

The nomain and nowalkback compilation switches must be used to compile external procedures for inclusion in resident libraries.

Now build the library. Normally you must run the Task Builder twice. The first run creates the loader map, which tells you the size of the library task image in words and in octal bytes. The second run actually builds the usable library.

The size of the task image is used to determine the base address of the library. The table on the next page simplifies this for you. The base address is the virtual address of the library and points to the highest multiple of 4K-word pages that can contain the library code. The base address and size in octal bytes are entered on the PAR option for the second run. The PAR option sets aside a partition for the library starting at a specified base address. The amount of memory reserved by the Task Builder depends on the size of the library. Of course, if you already know the size of the library, skip the first run and build the library using the PAR option and the appropriate values. The second TKB command is discussed on the next page.

Build the library procedure the first time, using the following command and options. The PAR option is not used with this build.

```
>TKB
TKB>SUM/-HD,SUM=SUM,LB:[1,1]PASLIB/LB  ——— /-HD is required
TKB>/
ENTER OPTIONS:
TKB>STACK=0 ——————— STACK=0 option is required
TKB>// ——————— notice the omission of the PAR option
```

The Task Builder has created two files: SUM.TSK, the executable library; and SUM.MAP, the load map showing, among other attributes, the size of the task image and the task address limits.

The two Task Builder options used in this command are /-HD and STACK=0. Both are required for building resident libraries. The /-HD option instructs the Task Builder to omit a task header from the task image. The STACK=0 option indicates that stack space is unnecessary for the resident library; a stack, instead, exists in each main task that shares this library.

Examine SUM.MAP to determine the size of the library task image in words and in octal bytes. The sixth line of the map file contains the size of the library in words. The seventh line contains the minimum and maximum task address limits (in octal) of the library. The minimum address limit is usually 0; the maximum address limit is the same as the size of the library in octal bytes (if the minimum address limit is 0). The size in octal bytes is simply the size of the library in decimal words, doubled and converted to octal.

Example: Memory Map Showing Task Size and Address

```
SUM.TSK   Memory allocation map  TKB M40.02   Page 1          19-AUG-83


Partition name  : GEN
Identification  : V2.1
Task  UIC       : [2,56]
Task attributes: -HD
Total address windows: 1.
Task  image  size  : 1408. words————————— size of the library in words
Task address limits: 000000 005353
          :
          :
          :————————————————————— the rest of the TKB map
```

From this map file you determine that the size of the library SUM.TSK is 1408 words or 5354 octal bytes.

Using the task image size and the table below, determine the base address of the library. The first column is the size of the task image in words, which was obtained from the map listing. The second column is the address limits of the task, in octal bytes. The octal value should lie within the address limit range corresponding to the task image size.

The third column is the base address of the resident library for a given task image size.

Image Sizes and Address Limits For Tasks

Task Image Size	Task Address Limits	Base Address
0. - 4095.	000000 - 017777	160000
4096. - 8191.	020000 - 037777	140000
8192. - 12287.	040000 - 057777	120000
12288. - 16383.	060000 - 077777	100000
16384. - 20479.	100000 - 117777	060000
20480. - 24575.	120000 - 137777	040000
24576. - 28671.	140000 - 157777	020000
28672. - 32767.	160000 - 177777	Can't create library

In this instance, with a library 1408 words long, SUM.TSK requires a base address of 160000_8 and is placed in the last active page (APR) by the Task Builder. The base address and the size in octal bytes (5354_8) are to be entered as **PAR** parameters for the second task build. Normally the size in octal bytes does not end on a 64-byte boundary (ending with two zeroes as in 6500_8), which the Task Builder requires. If you enter an odd size, the Task Builder issues the error message, "Illegal partition/common block specified," and waits for you to enter the **PAR** option again. Here the octal size 5354 should be rounded up to 5400 and entered on the **PAR** option, as shown:

```
>PAR=SUM:160000:5400
```

where **SUM** is the name of the library, **160000** is the base address (in octal) and **5400** is the rounded size of the library in octal bytes.

Task-build the resident library a second time to actually create the useable library. You must specify a file name in the third position on the command line to create the necessary .STB symbol table file associated with the library. In this case, SUM.STB defines the locations of the entry points of the library **SUM** so the Task Builder can map the library into the calling program's ~~virtual~~ address space. Here you must use the **PAR** option formatted above to identify the partition for the library.

```
>TKB
TKB>SUM/-HD,SUM,SUM=SUM,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=SUM:160000:5400 ——— the second build, with PAR
TKB>//
```

NOTE

Resident libraries used by Pascal programs should not be built position-independent; an absolute base address must be specified. Depending on the size of a main program, a floating base address can cause the Task Builder to load the resident library in the next available APR. As a result, there may not be enough heap space available to the Pascal task to allow it to expand when new is called. If the main program is executed and demands a large amount of dynamic memory, it may grow into the resident library area, resulting in the run-time error "Memory protection violation."

Create a partition on your system called SUM. You may wish to ask your system manager for help. Then install the task using the INS command, making it known to the system.

```
>INS SUM
```

To use this special library, you must edit the original program to remove the definition for the shared code (the procedure Sum) and substitute in its place a reference to the external procedure. The resulting program looks like:

```
program Wrtsum;

  procedure Sum(A,B: integer);————————— replaces code of procedure Sum
    external;

begin    { main program }
  Sum(1,5);
end.
```

This program is then compiled as any other Pascal program.

```
>PAS WRTSUM
```

Now, use the Task Builder to link this compiled Pascal program with the library you created above.

```
>TKB
TKB>WRTSUM/FP/CP,WRTSUM=WRTSUM,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESLIB=SUM/RO
TKB>//
```

The Pascal support library PASLIB is listed in the first Task Builder command line because, even though you are using a resident library, modules that are not in the resident library (such as $DATA) are still be loaded from PASLIB. The RESLIB option describes the location of the symbol table for the resident library. You could use the LIBR option if you build the resident library and its symbol table in the system UFD (LB:[1,1]). The /RO modifier indicates that the library is mapped as read-only.

To run the program, give the command:

```
>RUN WRTSUM
The sum is:        6
```

The program WRTSUM calls the procedure Sum in the resident library to print the result. Other users running WRTSUM (or calling Sum) at the same time would be sharing the code for the procedure Sum.

Converting an Entire Program to a Shared Procedure

The steps you take to convert an entire program into a shared procedure are identical to those described above for a single procedure. Before you try to convert all the procedures in a program to a resident library, be sure every procedure you intend to include in the library can be shared (i.e., those that do not create read/write data areas).

This example shows in general the way to convert an existing program into a shared procedure. The skeletal structure of a program, CH.PAS, looks like:

```
program Ch;

  { Global "const," "type," and "var" definitions }

    procedure A;
      { Definitions local to A }
      begin
        { procedure A }
      end;

  begin
    { Main program }
  end.
```

Since main programs are not permitted in a resident library, the program must be converted to a procedure that can be placed in a resident library. To convert a program to a procedure, change the program statement to an external procedure definition, and change the 'end.' of the main program to 'end;' to show that the program is now an external procedure. As a matter of course, global variables in the main program become local variables in the shared procedure. The code below has been changed in this way.

```
procedure Ch(var input, output: text);
  external;

procedure Ch;

  { Local "const," "type," and "var," definitions }

    procedure A;
      begin
        { Procedure A }
      end;

  begin
    { Procedure Ch }
  end;
```

The procedure is then compiled, task-built and installed as described earlier. At this point the resident library CH can be linked with other programs that call it. You now need to create a dummy main program that calls Ch, passing the standard files input and output, as shown in the file CHMAIN.PAS, below:

```
program ChMain;

  procedure Ch(var input, output: text);
    external;

  begin
    Ch(input,output);
  end.
```

CHMAIN calls the procedure Ch to do all of its processing. The task CHMAIN is relatively small because it contains only the data areas for the program. If several users run CHMAIN (or call Ch) simultaneously, each user shares the code (Ch) but retains an autonomous data areas.

Compile this program as you would any other main program. To build the main program and link it with resident library CH, use the following command and options.

```
>TKB
TKB>CHMAIN/FP/CP,CHMAIN=CHMAIN,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESLIB=CH/RO
TKB>//
```

The RESLIB option instructs the Task Builder to read the user-library definitions from the symbol table file CH.STB. If CH.STB were located in LB:[1,1], then the option LIBR=CH:RO could have been used because libraries in LB:[1,1] are considered system libraries. The RO switch requests read-only access to the resident library.

Read the sections about resident libraries and shared regions in the *RSX-11M/M-PLUS Task Builder Manual* before you attempt to create your own libraries. You may need to make changes to the library procedures, depending on the size of the library you create. In particular, you must change the PAR option, which is used to describe the virtual address of the library, as the library grows in size.

Compiler Optimizations

The Pascal-2 compiler implements these optimizations:

Variable Assignments to Registers

The compiler permanently assigns up to three floating-point accumulators and two general registers to commonly used local variables in each block. The compiler assigns the registers to the variables that are the most often used. No register is assigned for variables passed to a procedure as a **var** parameter or referenced directly by a procedure local to the declaring procedure. In addition, this optimization is disabled for the main program if any external procedures are referenced, since the compiler cannot determine what variables may be used by such routines.

Assignment of Constants and Addresses to Registers

The compiler attempts to fill all registers with useful operands during compilation of a procedure, since operations on registers are faster and take less space than the corresponding operation performed in memory. Once a procedure is compiled, unused registers are filled with constant operands and addresses if such assignment saves space. This low-level optimization often results in a saving in execution time as well.

Constant Folding

The compiler directly evaluates (folds) simple arithmetic involving constant operands of the types **integer**, **char**, **real**, and **boolean**. The generated code contains the result rather than the expression. Set expressions and relational expressions are not folded.

Dead Code Elimination

If statements and **case** statements are optimized if the selection expression is constant. In this case only one path of execution is possible, and the compiler discards others. Knowledge of this optimization can lead to the writing of conditional code much like that available in some preprocessors. For example:

```
if Debugging then writeln(SomeUserValue);
```

No code for this statement is generated if the identifier **Debugging** is defined as a constant with the value **false**.

The **debug** compilation switch disables this optimization.

Boolean Expression Optimization

When appropriate, Pascal-2 uses the minimum number of operations necessary to compute the final value of operands in boolean expressions, thereby reducing the cost of evaluating individual boolean expressions. This method is known as a "short-circuit" evaluation.

The programmer must be careful not to assume that all operands of boolean operators are evaluated or that some may not be evaluated. (This optimization takes advantage of a provision in the Pascal standard that allows an implementation to evaluate only the necessary operands of a boolean expression.) Also, the order in which the operands are evaluated is unpredictable.

The **debug** compilation switch disables this optimizaton.

Expression Targeting

The compiler can determine from context where a particular expression result should be computed. For instance, procedure parameters can often be computed directly on the run-time stack, and at times, expressions on the right side of the assignment operator can be computed directly into the variable on the left side.

Common Subexpression Elimination

Multiple occurrences of the same expression are detected and simplified. Such optimization of redundant expressions is needed even though a programmer can often avoid writing such code by introducing auxiliary variables. For instance, this example:

```
writeln(I + 1, I + 1);
```

may be simplified to:

```
J := I + 1; writeln(J, J);
```

The simplification avoids the redundant computation. However, redundancy of the sort shown in the first example often leads to a more readable program. Also, certain classes of redundant expressions cannot be eliminated in the source program. For instance, array index calculations involve several underlying operations that are not reflected in the source code and therefore cannot be simplified by the programmer. Pascal-2 eliminates a wide class of common subexpressions, across statement boundaries as well as within simple expressions.

The **debug** compilation switch disables this optimization.

Common Branch Tail Elimination

In some cases the compiler generates several branches to the same location in the object program. At times the compiler can replace redundant instructions preceding one such branch instruction with a branch to a point in the generated code that executes the same instruction stream. This low-level optimization executes an extra branch instruction in order to save some space.

The **debug** compilation switch disables this optimization.

Array Index Simplification

Index expressions of the form [*variable* + *constant*] and [*variable* − *constant*] are partially computed. The addition or subtraction of the constant operand is folded into the value computed for the base of the array. This optimization is enabled only if array bounds checking is disabled and the array is unpacked.

Appendix A: Compilation Error Messages

'(' expected	Check parameter list syntax.
')' expected	Check parameter list syntax.
',' expected	Check parameter list syntax.
'..' expected	Check array specification.
':' expected	Check type or var specification.
':=' expected	Check for undefined procedure or missing colon.
';' expected after procedure body	Use semicolons to separate procedure declarations.
'=' expected	Check constant or type syntax.
'[' expected	Check array index specification.
']' expected	Check array index or set specification.
']' or ',' must follow index expression	Check array index specification.
A type identifier is not allowed here	The compiler encountered a bad structured constant or misplaced identifier.
Actual parameter type doesn't match formal parameter type	Parameters being passed to procedures must have the same type names as the declared (formal) parameters.
Ambiguous switch	The specified command-line switch name does not contain enough characters to distinguish it from switches with similar names.
Array exceeds addressable memory	
Array subscript out of range	
Assignment of file variables not allowed	An attempt has been made to assign an expression to a file variable or one file variable to another.
Assignment operands are of differing or incompatible types	Type mismatch — compare left and right sides of assignment statement for compatibility. Note that pointer types must point to identical data structures.
Assignment to constants not allowed	
Assignment value out of range	
Bad adjust offset value in procedure <*name*>/main program	A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C for more information.
Bad CASE label	Case labels and case selectors must be of the same type. A colon, erroneously placed after the keyword **otherwise**, can also cause this error.
Bad constant	
Bad IN operands	The left operand must be of a scalar type; the right operand must be of a compatible set type.

Bad ORIGIN value	Origin values are restricted to the I/O page (locations 0 to 1000 octal) or the system area (28K to 32K).
Bad file name syntax	Check command-line syntax.
Bad parameter element	The indicated parameter element was not followed by a ',' or a ')'.
Bad type syntax	
Badly formed expression	Check parentheses and operator placement.
BEGIN expected	The statement part of a block must start with begin. Modules with no main program require the nomain compilation switch.
Binary operator expected	Two operands must be separated by an operator. Also check for mismatched quotes
Block declarations are incorrectly ordered	The relaxed ordering of declarations is an extension to standard Pascal and may be used only for global declarations.
Block ended incorrectly	
Block must begin with LABEL, CONST, TYPE, VAR, PROCEDURE, FUNCTION, or BEGIN	
Boolean value expected	
Can't assign a real value to an integer variable (use TRUNC or ROUND)	
Can't pack unstructured or named type	
CASE label defined twice	
CASE label does not match selection expression type	
CASE label must be non-real scalar type	
CASE label type does not match tag field type	
CASE selection expression must be a non-real scalar type	
Code too complex in procedure *<name>*/main program	The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this section for more information.
Compiler writer error - please contact Oregon Software at (503) 245-2202	This indicates an internal compiler error — please save all listings and terminal output.
Conflicting switches specified	Certain switch combinations cannot be specified together on the compiler command line. The object switch conflicts with the macro switch. The debug switch conflicts with both the profile and errors switches; the profile switch conflicts with the errors switch.
Declaration terminated incorrectly	
Declared labels must be defined in procedure body	
DO expected	Check for, while or with statement syntax.
END expected	
Exponent must lie in range -38..38	
Expression type is incompatible with FOR index type	For statement index types must be non-real scalars.

External procedures/functions must be defined at outermost level External procedures may not be defined within other procedures.

Extra END following block - Check BEGIN ... END pairing

Extra procedures found after main program body This error occurs when more than one main program body (starting with a **program** statement) appears in the source file.

Extra statements found after end of program This error occurs when more than one main program body appears in the source file, or when the **nomain** compilation switch is used with a source file that contains a main program body.

Field variable expected for NEW Additional parameters to **new** must be tag-field constant values that identify the particular variant record being allocated.

File cannot contain a file component An element of a file cannot itself contain a file.

File names in RESET/REWRITE are non-standard This error is generated only when the **standard** compilation switch is enabled.

File variable expected The first parameter to **reset**, **rewrite**, **get**, **put**, and **seek** must be a file variable.

File variable or pointer variable expected The indicated caret (^) has been incorrectly placed after a variable that was neither a pointer nor a file.

Files must be passed as VAR parameters

FOR-loop control variable can only be a simple non-real scalar variable

FOR-loop control variable must be declared at this level A **for** statement control variable must be declared local to the block containing the **for** statement.

Format expression must be of type INTEGER Field-width specifications in **write** or **writeln** statements must be integers.

Forward procedure/function body is never defined

Forward type reference is never resolved The type referenced in a pointer type declaration is not defined by later declarations.

Function cannot be applied to an operand of this type A standard function has been passed a parameter of the wrong type (for example, **trunc**/**round** can only be applied to **real** types).

Function identifier is never assigned a value

Function name expected

Function result must be of scalar or pointer type Functions may not return structured types such as records and arrays. Use **var** parameters to do this.

Function result type cannot be duplicated in forward-declared function body The parameter list and result type are already specified by the forward declaration and may not be repeated. Instead, simply give the function name.

Identifier cannot be redefined or defined after use at this level The specified identifier is already defined in the current block and cannot be assigned a new meaning in the indicated block.

Identifier expected The indicated argument should be a variable, not a constant or expression.

Illegal character

Illegal comparison of record, array, file, or pointer values Pointer types may be compared only for equality; record, array, and file types may not be compared in any case except strings.

Illegal function assignment

Illegal subrange The lower bound of a subrange is required to be less than or equal to the upper bound.

Index expression type does not match array declaration

Index must be non-real scalar type

Index variable missing in this FOR statement

Integer label expected

Integer overflow or division by zero

Integers must lie in range -32767..32767

Internal temp error in procedure <*name*>/main program A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C for more information.

Label cannot be redefined at this level Labels may be redefined within nested procedures, but not at the same level.

Label defined twice

Label is target of illegal GOTO Branching into `if-then-else` or `case` statements is illegal.

Label must be declared in LABEL declaration

Label must be unsigned integer constant

Line too long The maximum input line length is 160 characters.

Listing requested but no file provided Check command-line syntax.

More than two output file specifications Check command-line syntax.

Must assign value before using variable The standard states that variables must be initialized before they are used.

Must use VAR parameters with NONPASCAL directive

The calling sequence for `nonpascal` procedures and functions accepts only call-by-reference parameters.

Need at least 1 digit after '.' or 'E' Check for proper real numeric format.

Need at least one value to WRITE

Need at least one variable to READ

No file in field Check command-line syntax.

No input file provided Check command-line syntax.

"NO" not allowed on this switch

No strict inclusion of sets allowed The operators '<' and '>' may not be applied to set operands. Instead, use '<=' or '>='.

Non-standard comment form, please use "{" or "(*" The comment form '/*', '*/' is not accepted by Pascal-2. The PASMAT utility automatically converts non-standard comments to the standard form.

Nonsense discovered after program end Extraneous characters are present in the input file after the proper end of the program.

Octal constant contains an illegal digit Octal constants cannot contain an 8 or a 9.

Octal constants are not standard Pascal. This message is issued only when the standard compilation switch is specified and the conventional octal form containing 'B' is used.

OF expected Check file or set declaration syntax, or case statement syntax.

Only 15 levels of nesting allowed The compiler's limit for procedure and function nesting has been exceeded.

Only functions can be called from expressions Procedures do not return a value and may not be called from within expressions.

Operand expected

Operands are of differing or incompatible type

Operator cannot be applied to these operand types Check the indicated expression for proper form and operand type compatibility. For example, characters may not be multiplied together.

OTHERWISE/ELSE clause in CASE not allowed `Otherwise` is an extension to standard Pascal. This message is issued when the standard compilation switch is specified.

Out of memory in procedure <name>/main program The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this section for more information.

Output requested but no file provided Check command-line syntax.

Packed array [1..n] of characters expected The file name arguments in `reset` and `rewrite` must be strings.

Parameter list cannot be duplicated in forward-declared procedure/function body The indicated statement should simply give the procedure name and no parameters.

Pointer variable expected

Procedure name expected

Procedures cannot be followed by type definition The relaxation of declaration ordering applies only to global declarations, and to declarations in inner blocks which precede procedure and function definitions. The indicated declaration section is improperly placed.

PROGRAM heading expected This error occurs only if the standard compilation switch is set.

Readln, writeln, eoln, and page must be applied to text file

Reassignment of FOR-loop control variable not allowed The control variable of a `for` statement may not be modified inside the body of the `for` statement.

Record identifier expected A `with` statement must specify a record variable.

Same switch used twice Check command line for duplicate switches.

Set is constructed of incompatible types

Set types must have a base in the range 0..255

Sets must be non-real scalar type	The indicated set definition contains an illegal component type.
Statement ended incorrectly	
String constants may not include line separator	A closing single quote (') is missing.
String of length zero	Strings must contain at least one character.
Tag does not appear in variant record label list	The tag field referred to does not exist.
Tag identifier already used in this record	Field identifiers within a record are required to be unique and may not be redefined within that record.
The divisor of a MOD must be greater than zero	
THEN expected	Check `if` statement form.
This function was declared as a forward procedure	Conflict between declaration and use of function identifier. Check previous declaration.
This parameter cannot be followed by a format expression	A format expression may appear only in calls to `write` and `writeln`.
This procedure was declared as a forward function	Conflict between declaration and use of procedure identifier. Check previous declaration.
This procedure/function name has been previously declared forward	A procedure cannot be both `forward` and `external`, or both `forward` and `nonpascal`.
TO or DOWNTO expected	Check `for` statement syntax.
Too few actual parameters	The indicated parameter list does not agree with the procedure or function parameter definition.
Too many actual parameters	The indicated parameter list does not agree with the procedure or function parameter definition.
Too many errors!	The compiler error table holds 50 error messages. Error processing is terminated. Correct earlier errors and recompile for further checking.
Too many external references in procedure *<name>*/main program	Programs are limited to 256 external procedure references. See Appendix C of this section for more information.
Too many forward references (only 50 allowed)	
Too many identifiers (only 1597 allowed)	
Too many keys in procedure *<name>*/main program	The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this section for more information.
Too many labels in procedure *<name>*/main program	The limit of 280 case labels has been exceeded in named body of code. Restructure the program to reduce its complexity. See Appendix C for more information.
Too many nested INCLUDE directives (only 8 allowed)	
Too many nodes in procedure *<name>*/main program	The named body of code is too large to be compiled. Restructure the program to reduce its complexity. See Appendix C for more information.
Too many Pascal labels in procedure *<name>*/main program	More than 32 statement labels have been declared in named body of code. Restructure the program to reduce its complexity. See Appendix C for more information.

Too many procedures (only 300 allowed) The number allowed may vary according to size of the computer used and the version of the Pascal–2 software.

Too many strings or identifiers Restructure the program to reduce its complexity.

Too much object code in procedure <*name*>/main program The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C for more information.

Two file names in one field Check the command line for missing '=' or ','.

Travrs build error in main program A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C for more information.

Travrs walk error in main program A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C for more information.

Type name expected The first parameter passed to the `loophole` function must be a type name.

Unary '+' or '-' cannot be applied to set operands

Undefined identifier

Undeleted temps in procedure <*name*>/main program A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C for more information.

Unexpected ')' – Check for matching parenthesis

Unexpected ELSE clause – Check preceding IF for extra ';'

Unknown directive The legal directives are `%include` and `%page`.

Unknown switch Check command line for error.

UNTIL expected Check `repeat` statement for proper form.

Use '.' after main program body The indicated terminator is missing from `end` statement.

Use ';' to separate declarations In addition to flagging the usual missing-semicolon errors, this message is issued when an illegal digit for the specified radix is found in the cross-hatch '#' format for constants.

Use ';' to separate statements

VAR parameters cannot be passed an expression or packed field A `var` parameter must be the name of a variable or a component of a data structure. If the parameter is a component of a data structure (`record` or `array`), the structure may not be packed.

Variable name expected

Variable of type array expected

Variable of type record expected

Variables of this type are not allowed in READ Scalar variables may not be used in either `read` or `write` to a text file. Only predefined types (except `boolean`) and strings may be read from a text file.

Variables of this type are not allowed in WRITE Scalar variables may not be used in either `read` or `write` to a text file. Only predefined types (except `boolean`) and strings may be read from a text file.

Variant label is undefined

Appendix B: Run-Time Error Messages

2	**Array subscript out of bounds**	An array index is outside of the limits established for the array in the `type` declaration that defines the array.
13	**Attempt to read past end of file**	An input operation was attempted on a file when `eof` is true. This is usually due to a logic error in the program and can often be solved installing checks for `eof`. This error can be trapped with the `noioerror` procedure.
26	**Attempted reference through NIL pointer**	A pointer variable was improperly used while its value was undefined or `nil`. This error could be the result of a pointer being disposed of before it is used, or of a value never being assigned to it. This could also occur if the pointer was created with `loophole` or `ref`. The `$nopointercheck` switch suppresses this error message.
40	**Can't delete file**	The specified file cannot be deleted. This error can be trapped with the `noioerror` procedure.
10	**Can't open file**	The file could not be opened for the reason identifed by the I/O error code. For input files, this error usually occurs if the file does not exist. You can trap this error by specifying and checking the fourth parameter on the `reset` or `rewrite` statement used to open the file.
41	**Can't rename file**	The file could not be renamed for the reason given by the I/O error code. This error can be trapped with the `noioerror` procedure.
29	**CASE selector matches no label**	A `case` selector expression has no matching `case` label. The `otherwise` clause can be used to detect this error. The `$norangecheck` switch disables the detection of this error.
25	**Compiler/library mismatch**	The compiler version used to compile the main program does not match the support library used when the task was built. This error could occur if a new version of Pascal-2 is installed on your system, and you attempt to build a task with a module compiled with an older version of the compiler. The solution here is to recompile all of your modules. This error could also happen if the compiler or library were updated independently. You might have to rebuild the compiler for your system.
30	**DISPOSE() of a NIL pointer**	The pointer value does not point into the heap memory pool. This error could occur if the pointer is not initialized (via `new`) or if the pointer was created with `loophole` or `ref`.
4	**Division by zero**	Division by zero is not defined.

18	**Double deallocation of dynamic memory**	The pointer variable points to an area of memory already available for reuse. Possibly the pointer was not initialized, or it was created with **loophole** or **ref**. Also, the heap may have been corrupted. This can happen if you make assignments using uninitialized pointers or if an **external** procedure is called and the number of parameters passed to the procedure differs from the procedure definition.
14	**Error reading file**	An I/O error was detected while your program was reading an input file. The I/O error code describes the exact cause of the error. This error is most often reported during a **read** or a **get** operation. This error can be trapped with the **noioerror** procedure.
15	**Error writing file**	An I/O error was detected while your program was writing an output file. The I/O error code describes the exact cause of the error. This error is most often reported during a **write** or **put** operation. This error can be trapped with the **noioerror** procedure.
7	**EXP() overflow**	The parameter passed to the **exp** routine would cause an overflow condition during the calculation of the **exp**. The maximum value permitted is approximately 88.
- -	**Fatal initialization error**	This error indicates that the Pascal support library could not properly initialize the program for the reason given. This error usually occurs when the program is too large.
36	**FIS exception**	An FIS floating-point exception was detected. This error will only happen on machines such as the PDP-11/40, which use the FIS-style floating-point hardware.
22	**File is not a random access file. Use /SEEK**	This error is caused by an attempt to use the **seek** procedure on a file that was not opened with the **/seek** I/O control switch. The **/seek** switch must be used with the **reset** or **rewrite** statement that opened the file. This error can also occur if you attempt to open a sequential device such as a terminal or printer using **/seek**.
31	**File is not an input file**	An input operation was attempted on a file that has not been prepared for reading by **reset**. Be sure to use the **/seek** I/O control switch when you are opening random access files.
32	**File is not an output file**	An output operation was attempted on a file that has not been prepared for writing by **rewrite**.
11	**File name syntax error**	The file name is not a valid file specification. Check the file specification for invalid characters or other garbage in the file name. You can trap this error by specifying the fourth parameter on the **reset** or **rewrite** statement used to open the file.
24	**File not open**	All files other than **input** and **output** must be opened with **reset** or **rewrite** before they can be accessed.

5	**Floating point format error**	The program attempted to read a `real` number from a `text` file, where the data in the file is not a valid `real` number. This error can be trapped with the `noioerror` procedure.
3	**Floating point overflow**	The result of a floating-point operation is too large to represent as a real number. The magnitude of the largest real number is approximately $1.7E+38$.
19	**Illegal value for integer**	The program attempted to read an integer value that lies outside the range −32767..32767. This error can be trapped with the `noioerror` procedure.
38	**LUN already in use**	This error is caused by the use of the `lun:n` I/O control switch in a `reset` or `rewrite` where the logical unit number *n* is already in use. You can trap this error by specifying the fourth parameter on the `reset` or `rewrite` statement that opens the file.
8	**LOG() of zero or a negative number**	Logarithms are only defined for positive values.
35	**Memory protection violation**	The program attempted to reference memory unavailable to the task, or it attempted to modify a common area or a resident library marked as read-only. This error is most often caused by the use of an uninitialized pointer.
- -	**Multiple errors detected. Program aborted.**	This error occurs when an error is detected while another error is being processed. Rather than printing a possibly infinite list of errors, the support library prints this special error message and terminates the program. This error can be caused when the support library code has been accidentally overwritten.
17	**NEW() of zero length**	This error usually indicates an internal error in the Pascal support library. It could also be caused by an incorrect call to the `p$inew` function.
0	**No FPP support. Re-Task Build with /FP**	All Pascal–2 programs should be task-built with the `/FP` Linker switch if your machine supports FPP floating-point hardware. This switch forces the executive to preserve the contents of the floating-point registers when the Pascal task checkpoints. This error is never generated on non-FPP machines, where the `/FP` switch is not needed to build Pascal tasks.
1	**Not enough memory. Try making task checkpointable or extend $$HEAP.**	Your program is too large. Try overlaying ~~the~~ Pascal support library. If that does not work, try cutting down the size of the global data area of your program. You may need to divide up your program code into overlays. You might try expanding the program section `$$HEAP` to fill up all of memory. This can sometimes solve memory fragmentation problems.
37	**Not enough memory for file buffer**	This error is detected when there is no room on the heap to allocate a record buffer for the file being opened. If you are using the `buff:n`, `var:n` or `fix:n` I/O control switches, try reducing the record size. This error could also be the result of a memory fragmentation problem. Try expanding the program section named `$$HEAP` to fill up all of memory. It may be necessary to overlay the Pascal support library or your program. You can trap this error by specifying the fourth parameter on the `reset` or `rewrite` statement that opens the file.

33	Odd address trap	This error is most often caused by a reference through an uninitialized pointer. However, other problems could cause this error. Check for the improper use of overlays or a mismatch between external procedure definitions and references.
39	PUT() not at end of file	A put operation is valid only at the end of a sequential file. If you wish to update a record in the middle of a file, you must use the seek I/O control switch on the reset or rewrite statement that opened the file. This error can be trapped with the noioerror procedure.
23	Reserved instruction execution	Several problems could cause this error. Check for the improper use of overlays or a mismatch between external procedure definitions and references. If this error happens on a statement involving real numbers, you may have configured your Pascal-2 system incorrectly.
21	SEEK() to record zero	The record numbers for random access files start with record number 1. This error can be trapped with the noioerror procedure.
20	Set element out of range	The program attempted to reference an element of a set that is outside the range of values permitted in the set. The valid range is 0..255.
6	SQRT() of a negative number	The square root of a negative number is undefined.
27	Stack overflow. Try expanding $$HEAP	Several problems could cause this error. The default size of the stack is 2K words. If you have recursive procedures or procedures with large amounts of local variables, you may need to increase the maximum size of the stack by expanding the size of the program section $$HEAP using the EXTSCT option of the Task Builder. You could also get this error if you are using too much heap space. Close unused files or dispose of unused memory allocated via new.
12	Too many files open. Task Build with more UNITS	Use the UNITS option in the Task Builder to increase the number of logical unit numbers (LUNs) that the task can use. The default value is six, but Pascal uses two of these for terminal I/O. If you are using the Debugger or have more than four files open, use the UNITS option to increase the number of LUNs to a higher number such as 20. You can trap this error by specifying the fourth parameter on the reset or rewrite statement that opened the file.
16	TRUNC/ROUND overflow	The result of a trunc or round operation is too large to be represented. Only real numbers in the range −32767.0 to 32768.0 may be converted to integers with the trunc or round functions.
34	Unexpected BPT, IOT, EMT, or TRAP instruction	The only time the compiler generates any of these instructions is when the Pascal Debugger is being used. Check to be sure that none of your modules have been compiled with the debug compilation switch. Other problems could cause this error, such as the improper use of overlays or a mismatch between external procedure definitions and references.

- -	**Unknown Pascal run-time error #*num***	This message indicates that the detected error has no corresponding error message text. This indicates an internal error in the support library. Contact Oregon Software or file a Trouble Report.
9	**Unrecognized file switch**	An I/O control switch specified on a `reset` or `rewrite` statement is unknown to the file system. Check the spelling of your file switches. You can trap this error by specifying the fourth parameter on the `reset` or `rewrite` statement that opened the file.
28	Variable subrange exceeded	The program attempted to assign a value to a variable that is outside the bounds of the subrange type. This error is often caused by uninitialized variables or the improper use of subrange definitions. The `$norangecheck` switch disables the detection of this error.

Appendix C: Compiler Errors

Overflow Errors

Very complex or very large programs may exceed the capacity of the Pascal-2 compiler. Overflow of this sort is reported directly to the terminal rather than to the listing or error file. The compiler reports the type of overflow along with the name of the procedure causing the problem. Overflow errors also may occur in the main program. The following list of error messages assumes that a procedure named `MuchTooComplicated` has caused an overflow:

```
Too many keys in procedure MuchTooComplicated
Out of memory in procedure MuchTooComplicated
Too many labels in procedure MuchTooComplicated
Too many nodes in procedure MuchTooComplicated
Code too complex in procedure MuchTooComplicated
Too much object code in procedure MuchTooComplicated
Too many Pascal labels in procedure MuchTooComplicated
Too many external references in procedure MuchTooComplicated
```

An overflow condition in the main program is reported as:

```
Too many keys in main program
```

If compilation of a program causes one of the above error conditions, simplify the offending procedure or main program section. Two ways to do this are to split the routine into several sub-procedures and/or reduce the number of type definitions.

Consistency Checks

In addition to the above error messages, consistency checks within the compiler can trigger one of these errors:

```
Undeleted temps in main program
Internal temp error in main program
Travrs build error in main program
Travrs walk error in main program
Bad adjust offset value in main program
nnn consistency checks detected
Compiler writer error
```

You should seldom, if ever, see consistency-check errors; they are documented here for the sake of completeness. If you do see such an error, please send a Trouble Report to your software vendor immediately. Along with the Trouble Report, send the smallest possible source program that reproduces the error. Programs longer than one page should be sent on floppy disk or magnetic tape. (You also may call about the problem, if your support contract covers telephone response, but we undoubtedly need to have the problem in writing in order to fix it.)

Appendix D: Default File Extensions

The default file extensions listed here apply to files generated by and/or referenced by Pascal-2 and its utilities. The first column lists the type of file. The second column lists the extension.

File	Extension
Document	.DOC
Executable	.TSK
Library Symbol Table	.STB
Listing	.LST
MACRO-11 Source Code	.MAC
Object	.OBJ
Object Library	.OLB
Overlay Description	.ODL
PROCREF Output	.PRF
Profiler Output	.PRO
PROSE Input	.PRS
Source	.PAS
Symbol Map	.SMP
Symbol Table	.SYM
Temporary	.TMP
XREF Output	.CRF

Appendix E: Entry Points in the Pascal Support Library

Entry Point	Module	Source	Description
P$0	$INPUT	OPINP.MAC	Read character from standard file input
P$1	$DDIV	OPDDIV.MAC	Double-precision division simulation
P$2	$INPUT	OPINP.MAC	Read character from file
P$3	$DMUL	OPDMUL.MAC	Double-precision multiplication simulation
P$4	$RDINT	OPRDI.MAC	Read integer from standard file input
P$5	$DADD	OPDADD.MAC	Double-precision subtraction simulation
P$6	$RDINT	OPRDI.MAC	Read integer from text file
P$7	$DADD	OPDADD.MAC	Double-precision real addition simulation
P$8	$RREAL	OPRDR.MAC	Read real number from standard file input
P$9	$RREAL	OPRDR.MAC	Read double-precision real from standard file input
P$10	$RREAL	OPRDR.MAC	Read real number from text file
P$11	$RREAD	OPRDR.MAC	Read double-precision real from text file
P$12	$RDSTR	OPRST.MAC	Read string from standard file input
P$13			Permanently undefined (unlucky)
P$14	$RDSTR	OPRST.MAC	Read string from text file
P$15			Reserved
P$16	$INPUT	OPINP.MAC	Readln on standard file input
P$17			Reserved
P$18	$INPUT	OPINP.MAC	Readln from text file
P$19			Reserved
P$20	$OUTPT	OPOUT.MAC	Write character to standard file output
P$21			Reserved
P$22	$OUTPT	OPOUT.MAC	Write character to text file
P$23			Reserved
P$24	$WRINT	OPWRI.MAC	Write integer to standard file output
P$25			Reserved
P$26	$WRINT	OPWRI.MAC	Write integer to text file
P$27			Reserved
P$28	$WREAL	OPWRR.MAC	Write real number to standard file output
P$29	$WREAL	OPWRR.MAC	Write double-precision real to standard file output
P$30	$WREAL	OPWRR.MAC	Write real number to text file
P$31	$WREAL	OPWRR.MAC	Write double-precision real to text file
P$32	$OUTPT	OPOUT.MAC	Write string to standard file output
P$33	$INIO	OPINIO.MAC	Initialize standard files input and output
P$34	$OUTPT	OPOUT.MAC	Write string to text file
P$35	$IOERR	OPIOER.MAC	Set user-handling of I/O errors for file (noioerror)
P$36	$OUTPT	OPOUT.MAC	Writeln to standard file output
P$37	$IOERR	OPIOER.MAC	Status check of last file operation (ioerror)
P$38	$OUTPT	OPOUT.MAC	Writeln to text file
P$39	$IOERR	OPIOER.MAC	I/O error code of last file operation (iostatus)
P$40			Reserved

Entry Point	Module	Source	Description
P$41	$P2ERR	OP2ERR.MAC	"Stack overflow" error message
P$42			Reserved
P$43	$P2ERR	OP2ERR.MAC	"Subscript out of bounds" error message
P$44			Reserved
P$45	$P2ERR	OP2ERR.MAC	"Variable subrange exceeded" error message
P$46			Reserved
P$47	P2ERR	OP2ERR.MAC	"Reference through a nil pointer" error message
P$48			Reserved
P$49	P2ERR	OP2ERR.MAC	"Case selector" error message
P$50			Reserved
P$51	$P2ERR	OP2ERR.MAC	"Division by zero" error message
P$52			Reserved
P$53	$DELF	OPDEL.MAC	Delete file
P$54			Reserved
P$55	$RENAM	OPREN.MAC	Rename file
P$56			Reserved
P$57	$CLOSE	OPCLOS	Close files in specified range
P$58			Reserved
P$59	$DATA	OPDATA.MAC	Initialize Pascal
P$60	$PUT	OPPUT.MAC	Put next record
P$61	$INPUT	OPINP.MAC	Get next record
P$62	$IO	OPIO.MAC	Break file
P$63	$END	OPEND.MAC	Program termination
P$64	$OPEN	OPOPEN	Rewrite file
P$65	$SEEK	OPSEEK.MAC	Seek record in file
P$66	$OPEN	OPOPEN.MAC	Reset file
P$68	$CLOSE	OPCLOS.MAC	Close file
P$70	$DYNMM	OPDYN.MAC	New memory allocation
P$72	$DYNMM	OPDYN.MAC	Dispose memory deallocation
P$74			Reserved
P$75	$REG	OPREG.MAC	Save registers
P$76			Reserved
P$77	$REG	OPREG.MAC	Restore registers
P$78	$ARITH	OPINT.MAC	Signed integer multiply
P$79			Reserved
P$80	$ARITH	OPINT.MAC	Signed integer divide
P$81	OPPACK	OPPACK.PAS	Pack
P$82	$ARITH	OPINT.MAC	Signed integer mod
P$83	OPPACK	OPPACK.PAS	Unpack
P$84	$FCMP	OPFCMP.MAC	Floating compare simulation
P$85	$DCMP	OPDCMP.MAC	Double-precision floating compare simulation
P$86	$CNVRT	OPCNV.MAC	Trunc of real number
P$87	$CNVRT	OPCNV.MAC	Trunc of double-precision real
P$88	$CNVRT	OPCNV.MAC	Float conversion to real
P$89	$CNVRT	OPCNV.MAC	Float conversion to double-precision real
P$90	$FSQRT	OPFSQR.MAC	Sqrt of real number

Entry Point	Module	Source	Description
P$91	$DSQRT	OPDSQR.MAC	Sqrt of double-precision real
P$92	$FTRIG	OPFTRG.MAC	Sin of real number
P$93	$DTRIG	OPDTRG.MAC	Sin of double-precision real
P$94	$FTRIG	OPFTRG.MAC	Cos of real number
P$95	$DTRIG	OPDTRG.MAC	Cos of double-precision real
P$96	$FATN	OPFATN.MAC	Atn (arctangent) of real number
P$97	$DATN	OPDATN.MAC	Atn of double-precision real
P$98	$FEXP	OPFEXP.MAC	Exp (exponential) of real number
P$99	$DEXP	OPDEXP.MAC	Exp of double-precision real
P$100			Reserved
P$101			Reserved
P$102	$FLOG	OPFLOG.MAC	Ln (natural logarithm) of real number
P$103	$DLOG	OPDLOG.MAC	Ln of double-precision real
P$104			Reserved
P$105			Reserved
P$106	$FTIME	OPFTIM.MAC	Time function -real
P$107	$DTIME	OPDTIM.MAC	Time function - double-precision real
P$108	$CNVRT	OPCNV.MAC	Round of real number
P$109	$CNVRT	OPCNV.MAC	Round of double-precision real
P$110	$WBOOL	OPBOOL.MAC	Write boolean to standard file output
P$111	$FORT	OPFORT.MAC	Fortran interface
P$112	$WBOOL	OPBOOL.MAC	Write boolean to text file
P$113	$ERR	OPERR.MAC	Error reporting
P$114			Reserved
P$115			Reserved
P$116	$ARITH	OPINT.MAC	Unsigned integer multiplication simulation
P$117	$FSIM	OPFSIM.MAC	Real division simulation
P$118	$ARITH	OPINT.MAC	Unsigned integer division simulation
P$119	$FSIM	OPFSIM.MAC	Real multiplication simulation
P$120	$ARITH	OPINT.MAC	Unsigned integer mod
P$121	$FSIM	OPFSIM.MAC	Real subtraction simulation
P$122			Reserved
P$123	$FSIM	OPFSIM.MAC	Real addition simulation
P$124			Reserved
P$125			Reserved
P$126			Reserved
P$127	$REG	OPREG.MAC	Check for stack overflow
P$128			Reserved
P$129			Reserved
P$130			Reserved
P$131			Reserved
P$132			Reserved
P$133			Reserved
P$134			Reserved
P$135			Reserved

Pascal-2 V2.1/RSX Language Specification

Introduction to the Language Specification

The Pascal-2 compiler processes the standard Pascal language, described by International Standards Organization publication *ISO 7185*, published in late 1983, which we call the "standard" hereafter. The ISO standard is identical to the British standard (BS 6192). The ISO standard allows two levels of conformance, Level 1 (including conformant array parameters) and Level 0 (not including conformant array parameters). The American ANSI-IEEE standard is identical to Level 0 of the ISO standard.

Compliance is Level 1: conformant array parameters are included. Pascal-2 includes the extensions detailed in this guide. This guide includes data on non-standard language features. This guide is not intended as a full language document.

Syntax definitions in this specification use the notation described in Appendix C, Pascal-2 Syntax.

Changes in the Standard

Because you may not be familiar with all the changes to the Pascal language from Jensen and Wirth (1978) to the standard as adopted in 1983, this section outlines those changes and Pascal-2's method of implementing them.

'For' Statement Control Variables

Variables that control a `for` statement must be simple variables, local to the routine in which the `for` statement is written. Originally, any variable could be used.

File Declaration

The standard states that the files `input` and `output` are automatically declared as global variables if they are mentioned in the program heading. Because program headings are optional in Pascal-2, `input` and `output` are declared as global variables in every Pascal-2 program. Thus, you cannot redefine `input` or `output` at the global level. In earlier versions of the language, the actual point of definition was undefined.

Parameter Compatibility

The compatibility rules for `var` parameters are now defined according to a restrictive rule, which requires the argument passed to have the same type as the formal parameter. Although the types must be the same, the type identifiers may differ. The appearance of a new type construct creates a new type. Previously, the rules for `var` parameters were undefined.

Procedure and Function Parameters

The standard has changed the method of declaring procedure and function parameters. The new syntax provides a way of checking the parameters of these procedures and functions, thus reducing the likelihood of type errors.

The syntax for a parameter list is:

parameter-list = "(" *parameter-section*
    { ";" *parameter-section* } ")" .

parameter-section = (["var"] *identifier* { "," *identifier* } ":"
    (*identifier* | *conformant-array-schema*))
    | *procedure-heading* | *function-heading* .

A full procedure heading must be provided for any procedure or function declared as a parameter, and the procedure heading for any procedure or function passed as an actual parameter must match. For example:

```
var
  K, L: integer;

  procedure P(procedure Q(I, J:integer));
  begin
    Q(K, L);
  end;

  procedure P1(I, J: integer);
  begin
    writeln('test of proc parameters', I, J);
  end;

begin
  K := 1;
  L := 20;
  P(P1);
end.
```

The program issues the following output:

```
test of proc parameters        1       20
```

The standard does not allow a standard function to be used as a parameter for a function or procedure. To pass a standard function as a function or procedure argument, you must define a function that calls the standard function, then pass the user-defined function as the function or procedure argument.

Conformant Array Parameters

Normally, a procedure or function accepts an array parameter containing a fixed number of elements. The number of elements holding meaningful information may vary but the size of the array may not. If you need to pass arrays of different lengths, you have to declare and pass a general array that is as long as the longest possible array, and you must track the last element of each. Another approach is to write a separate procedure to handle each size of array, which is clearly inefficient.

Use of conformant array parameters solves this problem. Conformant array parameters are formal parameters that allow you to write a general procedure or function that, at each activation, accepts array parameters of different size and with different lower and upper bounds. At activation, the upper and lower bounds of the conformant array parameter assume the upper and lower bounds of the passed parameter (the actual parameter).

The syntax for a conformant array parameter is:

conformant-array-parameter-specification =
        ["**var**"] *identifier-list* ":" *conformant-array-schema* .

conformant-array-schema = *packed-conformant-array-schema*
        | *unpacked-conformant-array-schema* .

packed-conformant-array-schema = "**packed array**"
        "[" *index-type-specification* "]" "**of**" *type-identifier* .

unpacked-conformant-array-schema =
        "**array**" "[" *index-type-specification*
        { ";" *index-type-specification* } "]" "**of**"
        (*type-identifier* | *conformant-array-schema*) .

index-type-specification = *bound-identifier* ".." *bound-identifier* ":"
        type-identifier .

As the EBNF diagrams show, a conformant array schema may be either packed or unpacked. An unpacked conformant array may be nested within itself or within other conformant arrays (either packed or unpacked); if so, an abbreviated form may be used. In the example below, Mx is the conformant array parameter being used in Examp. T1, T2 and T3 are data types. The two definitions are equivalent. Notice that the semicolon in the abbreviated form replaces '] of array [' in the long form.

Example: Full and Abbreviated Forms of Conformant Array Parameters

```
procedure Examp(var Mx: array [Lb1..Ub1: T1] of array [Lb2..Ub2: T2] of T3);
```
or
```
procedure Examp(var Mx: array [Lb1..Ub1: T1; Lb2..Ub2: T2] of T3);
```

An array may be passed as a conformant array parameter if:

- the elements have the same types,
- the index types are compatible, and
- the bounds are within the range specified by the parameter declaration.

If two parameters are specified with a single conformant array schema, the actual parameter passed must have the same type. Also, a value conformant array may not be passed as a parameter to another procedure or function.

The next example demonstrates the use of conformant array parameters. The formal parameter Arr is a conformant array parameter and takes the values of two different-sized arrays, First and Second. At the first activation of the function AddArray, the two elements of array First are added together to reach a sum. The next activation adds up the four elements of array Second and arrives at a different sum, as shown in the output following the program listing.

Example: Use of Conformant Arrays

```
program Conform;
  var
    First: array [1..2] of integer;      { two-element array  }
    Second: array [0..3] of integer;     { four-element array }
    Total: integer;

  function AddArray(var Arr: array [Lower..Upper: integer] of integer): integer;
    var
      I, Sum: integer;
    begin
      Sum := 0;
      for I := Lower to Upper do
        Sum := Sum + Arr[I];
      AddArray := Sum
    end;

  begin
    First[1] := 5;  First[2] := 9;
    Total := AddArray(First); ———————— called with two-element array
    writeln('Total for this array is: ', Total:5);
    Second[0] := 1;  Second[1] := -31;  Second[2] := 77;  Second[3] := 15;
    Total := AddArray(Second);-———————— called with four-element array
    writeln('Total for this array is: ', Total:5)
  end.
```

Running the program yields:

```
Total for this array is:     14———————— sum of elements of array First
Total for this array is:     62———————— sum of elements of array Second
```

For a practical example of the use of conformant array parameters, examine the source code of Pascal–2's Dynamic String Package, L1STRING, one of the utilities.

Literal Strings

A literal string may not extend over more than a single line. Earlier standards were unclear on this point. The limitation allows better diagnostics for unterminated strings.

'Write,' 'Writeln' of 'Packed Array of Char'

A **write** or **writeln** procedure call applied to a packed array of char writes only as many characters as the field-width parameter specifies. If the packed array of char exceeds the field-width, the string is truncated. The string is right-justified if the specified field width is longer than the packed array. If no field width is specified, a **write** or **writeln** writes as many characters as are in the string.

Example: Truncation of Sring that Exceeds Field Width

```
program Buff;
  var
    Buffer: packed array [1..30] of char;
    BuffCount: integer;

  begin
    Buffer:= 'This is a packed array of char';
    writeln(Buffer);
    BuffCount := 6;
    writeln(Buffer:BuffCount);
    writeln('cutoff':3);
    write('shorter':10);
  end.
```

When executed, the program yields these results:

```
This is a packed array of char
This i
cut
   shorter ──────────────── note leading blanks
```

Identifiers

The initial character of an identifier must be an alphabetic character or a dollar sign. All other characters making up identifiers may be any combination of digits, letters, dollar signs or underbars. Identifiers may be of any length; all characters are significant. Lower-case characters are interpreted in the same way as upper-case characters. For example, **name**, **Name**, **NamE**, and **NAME** are equivalent. See "Syntax Extensions" for details on the the use of the non-standard dollar sign and underbar in identifiers.

Alternate Symbol Representations

The standard now defines alternate representations for symbols that are unavailable in some character sets. These symbols are equivalent:

Alternate Symbols

Standard	Alternate Symbol	Future
^ or ↑	@ ('at' sign)	
{	(*	
}	*)	
[	(.	
]	.)	

The alternate comment delimiters are equivalent to the standard comment delimiters, and a comment may open with one type of delimiter and close with the other. Comments may not be nested.

Example: Valid and Invalid Comments

```
(* This is a valid comment }
{ This is (* not *) a valid comment }
{ This { { { is a valid comment. }
```

Any additional opening comment delimiters that appear after the first opening comment delimiter and before the closing comment delimiter are treated as text and have no effect on the comment.

Implementation Definitions

This section provides details and characteristics of implementation-defined elements of Pascal–2.

Standard Type 'Integer'

The predefined identifier **maxint** has the value 32767.

The standard type **integer** has the range (-32767..32767). An unsigned (extended-range) integer may be defined with the range 0..65535. See "Unsigned Integer Conversion" in the Programmer's Reference.

Standard Type 'Real'

A **real** variable has the standard PDP-11 single-precision or double-precision floating-point structure, with magnitude in the range 1E-38..1E+38. Single-precision values give approximately 7 decimal digit precision; extended (double-precision) values give approximately 15-digit precision. Arithmetic overflow is detected for all **real** operations, but underflow is ignored and returns a result of zero.

The standard transcendental routines are accurate to 6 decimal digits in single precision and to 15 decimal digits in extended precision.

Standard Type 'Char'

The standard does not define the character set to be used internally to represent **char**. Pascal–2 uses 8-bit characters, allowing the use of the extended version of the ASCII character set, rather than 7-bit characters to represent the standard ASCII character set. The most significant bit is "off" unless used with extended character sets. **Ord(char)** is in the range 0..255.

Programs that calculate bit or byte offsets into a packed structure should treat a character as 8 bits, not 7; and storage size is the same for characters in either packed or unpacked structures.

Standard Type 'Text'

The standard type **text** is a file type with components of type **char**. **Text** is implemented as a file of variable-length records containing ASCII data. If a new file is being created, the maximum record length is assumed to be 132 characters unless set to some other value by the **var:n** switch in the **rewrite** call. See "I/O Support Extensions" for details. If the program attempts to generate a record longer than the maximum length, Pascal–2 breaks up the record as though a **writeln** had been called.

'Set' Types

Pascal–2 limits a **set** to a maximum of 256 elements. The lower and upper bounds must lie in the range 0..255, e.g., **set of 4..9**. The declaration **set of integer** is equivalent to the declaration **set of 0..255**. See "Undetected Errors" for restrictions on the checking of integer sets.

I/O Definitions

The following table summarizes the default field widths used when values are written to a text file:

Default Field Widths

Value Type	Field Width
integer	7
real	13
boolean	5

The floating-point representation of a real number includes the sign of the number (a space for positive numbers and a '-' for negative numbers), the real number in scientific notation, an upper-case `E` signifying exponential notation, the sign of the exponent ('+' or '-'), and a two-digit exponent. For example, the real number −105.39 prints as `-1.053900E+02`.

Boolean values are written in upper case (`TRUE`, `FALSE`). In the five-character default field, the value `TRUE` is right-justified, with a leading blank before the 'T'.

The procedure `page(F)` inserts a form feed (page eject) into the file specified by the required file argument. Calling `page(F)` with data in the file buffer executes `writeln(F)`, which writes the remainder of the buffer, and `write(F,chr(12))`, which writes the form-feed character. Calling `page(F)` with an empty file buffer results in a page eject only.

If associated with the standard input file (the terminal), `reset(input)` performs the equivalent of a `readln`, but otherwise has no effect; in the same way, `rewrite(output)` prints any incomplete line, but otherwise has no effect. `Reset(output)` or `rewrite(input)` produces an error message. See "External File Access" for details on the use of the extended form of `reset(input)` or `rewrite(output)`.

Syntax Extensions

This section describes Pascal-2 extensions to the syntax of standard Pascal.

Identifiers

In standard Pascal, the first character of an identifier name must be an alphabetic character, with the remaining characters alphanumeric.

In Pascal-2, the character $ (dollar sign) is allowed in an identifier anywhere an alphabetic character is allowed. The character _ (underbar) is allowed anywhere a numeric character is allowed. For example, the identifier _ABC is not valid because it begins with an underbar. The following are legal Pascal-2 identifiers:

```
system$name
$$file
this_is_a_long_identifier
This___Is_Also___Legal
```

Program Heading

In standard Pascal, the program heading is required, and the parameters define the external files to be used:

```
program Test (input, output, File3);
```

In Pascal-2, the program heading and parameters are not required. If present, they are checked for proper syntax. The file parameters are otherwise be ignored. **Input** and **output** are automatically declared file variables. Every other external file must be specified by an additional parameter allowed in the standard procedures **reset** and **rewrite**. See "External File Access" under "I/O Support Extensions" for details.

Though not required, inclusion of the program name on the program statement is still a good practice because it names the object module for main programs and external modules. Further, the program name is used to name the psect when the **own** compilation switch is specified.

Declaration Order

The declaration sections **label**, **const**, **type**, **var**, **procedure**, and **function** may be interleaved as desired at the global level of a program. **Const** and **type** may be interleaved at other levels. This extension is useful for source module inclusion and structured constant definitions as described below. Any number of declaration sections of each type may be present. An identifier still must be defined before the identifier is used in any other way.

'%Include' Lexical Directive

A special directive allows separate text files to be included within a program. The contents of the separate file are inserted into the program at whatever point the `%include` directive occurs. Included files may themselves contain `%include` directives, nested to a maximum of seven levels.

The syntax for the `%include` directive is:

`%include` *'file-name-string'*;

The *file-name-string* must contain at least the name of the file; if no file extension is specified, .PAS is assumed. In addition to the file name and extension, *file-name-string* may contain the disk volume number, the UIC and the version number of the file.

The single quotes (`'...'`) enclosing *file-name-string* are required if a file version is included as part of the string; otherwise they are optional. The quotes allow the compiler to distinguish the semicolon immediately preceding the version number from the semicolon that terminates the directive.

Example: File Names In An 'Include' Directive

```
%include hdr;
%include 'makhdr.pas';
%include term.doc;
%include '[2,16]rebld';
%include 'db0:[15,7]decl';
%include '[5,43]lib.pas;6';
```

See the Programmer Reference for details.

'%Page' Lexical Directive

The `%page` directive causes a page break (form feed) to occur in the listing file immediately following the line on which the `%page` directive is placed. The `%page` directive itself is printed in the listing file on the last listed line of the page preceding the page eject. The ending semicolon is optional.

'External' and 'NonPascal' Directives

Similar to the `forward` standard directive, the `external` directive distinguishes a particular Pascal procedure or function that is separate from the module that invokes it. An external procedure must be declared at the global level. If the body of an external procedure or function does not appear in a compilation, it is assumed that the body will be in another object module. If the body of the external procedure does appear, its name will be made available in the object module for reference by other modules. References to the external procedure are resolved at link time.

Limitations of the object module structure require that external names be distinct within the first six characters. The underbar cannot be expressed in the object module format and is replaced by a period in the external name. No type checking is done for parameters of an external routine.

The **nonpascal** directive may be used instead of **external** if the external procedure is written in a language other than Pascal. **Nonpascal** generates the Digital standard calling sequence used by FORTRAN and most MACRO-11 routines. This calling sequence uses register R5 as a pointer to a list of parameters. All parameters are passed by reference, so only **var** parameters may be used. (The Pascal calling sequence places parameters on the stack.) MACRO-11 routines written with the Pascal-2 PASMAC utility must be declared as **external** rather than **nonpascal**, because PASMAC simulates the Pascal calling sequence.

See also "External Modules" in the Programmer Reference for details.

Structured Constants

The syntax for constant definitions is extended to allow you to specify constants in record and array types. Under the standard, arrays or records cannot be assigned values in the program's *constant-definition-part* and each element must be assigned a value in the program body with an assignment statement. The structured-constants language extension eliminates the need to use assignment statements to assign values to constants of type **array** or **record**.

The formal syntax for structured constants is:

structured-constant =
    structured-type-identifier constant-component-list .

constant-component-list = "(" *constant-component* { "," *constant-component* } ")" .

constant-component = *constant* | *constant-component-list* .

where

structured-type-identifier Is a data type with an array or record structure. All of the components of that structure must be of simple types, array types, or record types.

constant-component-list Identifies each constant used as components between nested levels of parentheses. If an element is another structured type, a constant type of the same structure may appear or its elements may be set individually between inner parentheses.

constant-component Must correspond one to one with the component of the structured (array or record) type, and each *constant-component* must be a constant of the same type as the corresponding structure component. An access to the structure component returns the value of the *constant-component*. If the structure component is of a structured type, only the corresponding *constant-component-list* must be provided, declared with the proper syntax.

The following are valid declarations. Note that the data types needed by the structured constant must be declared before the structured constants.

```
type
  S1 = packed array [1..4] of char;
  S2 = record
    String: S1;
    end;

const
  C1 = S1('a', 'b', 'c', 'd');
  C2 = S2('abcd');
```

The *structured-type-identifier* for individual components need not be provided.

An optional record field specification, called a *variant*, may be used within a structured constant. The following is a valid declaration.

```
{$nomain}

type
  xtype = (a,b,c,d);
  S1 = packed array [1..4] of char;
  S2 =
    record
      case x: xtype of
        a:
          (string: S1);
        b:
          (i: integer);
        c:
          (j: real);
        d:
          ();
    end;

const
  C1 = S1('a', 'b', 'c', 'd');
  C2 = S2(a,'abcd');
  C3 = S2(b, 1);
  C4 = S2(c, 2.47);
  C5 = S2(d);
```

Variant records may or may not use a tag field, but a tag value must be provided in the *constant-component-list*.

Three examples are presented showing several uses of structured constants. The first example illustrates the nesting of parentheses in the structured constant declarations. The second example compares the standard's method of declaring record or array constants with the structured constant method. The third example shows the correct way to declare multidimensional arrays of constants.

Nested Parentheses

The structured constant Workers in the following declarations contains three levels of parentheses: the first for the array structure; the next for the outer record; the last for the pay information. The fourth constant in the array, however, for Maxine, contains a structured element that is set by reference to a constant of the same type with no further inner parentheses.

Example: Nested Parentheses

```
type
  Compensation = (Paid, Unpaid);
  Paytype = Record
    Title : (Clerk, Indian, Chief, President);
    case Compensation of—— note each variable and field appears
      Paid:    (Rate: real);
      Unpaid: ();
    end;
  Employeetable = array[1..4] of record
    Name : packed array[1..10] of char;
    Payinfo : Paytype;
    end;

const
  Conchief = Paytype(Chief, Paid, 6.85);
note redefinition for Maxine
  Workers = Employeetable(
    ('Charlie    ', (Clerk, Paid, 3.40)),
    ('Samuel     ', (Indian, Paid, 5.25)),
    ('Edward     ', (President, Unpaid)),
    ('Maxine     ', Conchief)-———— note condensed form
    );
```

Each new structure must be enclosed in a set of parentheses, as shown above.

Standard vs. Structured Constant Declarations

A comparison with the standard method of declaring constants for arrays and records illustrates the efficiency and ease of use of structured constants. The program used in the comparison, DAYCALC, calculates the day of the week for any date. The declarations below are those required to declare two arrays of constants, MonthName and DayOffset. Note that the type declarations are identical in both cases. The declaration of other data types, constants and variables have been omitted.

To conform to the standard, constants in arrays and records must be declared and assigned values as shown below. This method requires many more statements and probably takes longer to execute than the equivalent structured constants declarations, shown after the standard example.

Example: Use of Standard Constant Declarations

```
program DayCalc;
  type
    Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
             Unknown);
    Name = packed array [1..3] of char;
    NameList = array [Month] of Name;
    DayOffsetList = packed array [Month] of 0..6;

  var
    MonthName: NameList;          { Text for name of month }
    DayOffset: DayOffsetList;     { Day mod 7 }

  begin    { DayCalc }

    MonthName[Jan] := 'jan'; MonthName[Feb] := 'feb'; MonthName[Mar] := 'mar';
    MonthName[Apr] := 'apr'; MonthName[May] := 'may'; MonthName[Jun] := 'jun';
    MonthName[Jul] := 'jul'; MonthName[Aug] := 'aug'; MonthName[Sep] := 'sep';
    MonthName[Oct] := 'oct'; MonthName[Nov] := 'nov'; MonthName[Dec] := 'dec';
    MonthName[Unknown] := '???';

    DayOffset[Jan] := 0; DayOffset[Feb] := 3; DayOffset[Mar] := 3;
    DayOffset[Apr] := 6; DayOffset[May] := 1; DayOffset[Jun] := 4;
    DayOffset[Jul] := 6; DayOffset[Aug] := 2; DayOffset[Sep] := 5;
    DayOffset[Oct] := 0; DayOffset[Nov] := 3; DayOffset[Dec] := 5;
    DayOffset[Unknown] := 0;
             :
             :
             : ———————————————————— rest of program goes here
```

With structured constants, your code is shorter and easier to maintain than with the standard method; however, the program is non-standard and is not necessarily portable to other Pascal implementations.

Example: Use of Structured Constants

```
program DayCalc;
  type
    Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
             Unknown);
    DayOffsetList = packed array [Month] of 0..6;
    Name = packed array [1..3] of char;
    NameList = array [Month] of Name;

  const
    MonthName = NameList('jan', 'feb', 'mar', 'apr', 'may', 'jun',
                         'jul', 'aug', 'sep', 'oct', 'nov', 'dec', '???');
    DayOffset = DayOffsetList(0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5, 0);
              :
              :
              :———————— rest of declarations and program body goes here
```

Multidimensional Arrays of Constants

Multidimensional arrays of constants, such as the two-dimensional array below, may be declared as in the following program. This program prints the elements of the two-dimensional array Table in the order they are stored.

Example: Use of Two-Dimensional Array

```
program TwoDimensions;
  const
    MaxElem = 3;

  type
    CharTable = packed array [1..MaxElem, 1..MaxElem] of char;

  const
    Table = CharTable(('x', 'y', 'z'),
                      ('a', 'b', 'c'),
                      ('p', 'd', 'q'));
  var
    I, J: integer;

begin     { TwoDimensions }
  for I := 1 to MaxElem do
    for J := 1 to MaxElem do
      write(Table[I,J], '  ');
  writeln;
end.      { TwoDimensions }
```

Running this program yields:

```
x  y  z  a  b  c  p  d  q
```

Statement Labels

In standard Pascal, statement labels are limited to the range 0..9999, whereas Pascal–2 allows the use of labels in the range 0..32767.

Default Case Label ('Otherwise')

A default statement may be included in a case statement according to the following syntax:

case-statement = "case" *case-index* "of" [*case-element*{ ";" *case-element* }]
[";"] ["otherwise" *default-statement* [";"]] "end" .

The default statement, which immediately follows the **otherwise** clause, is executed if no case label matches the value of the *case-index*. A special note on **otherwise** syntax: in contrast to the case label, which requires a colon, the **otherwise** clause must not contain a colon or a compilation error results.

Example:

```
case I of
  1: Ch := ':';
  9: Ch := ';';
otherwise Ch := '*';
end;
```

The list *case-element* is optional, so the following example is valid.

```
case I of
otherwise I := 1;
end;
```

I/O Support Extensions

I/O support extensions provide the Pascal–2 programmer with additional control of the interface to the operating system.

External File Access

The standard procedures **reset**, for opening an existing file, and **rewrite**, for creating a new file, have been extended to accept optional arguments that give Pascal–2 programs the ability to associate internal file variables with external file or device specifications. The syntax is as follows:

```
rewrite(file-variable, device-or-file-name, default-values, file-status);
reset(file-variable, device-or-file-name, default-values, file-status);
```

where

file-variable is a standard Pascal file variable.

device-or-file-name specifies the name of an external file with which the file variable is to be associated. This parameter, which may be a device or file name specification, must be a string type and may be either a literal string or a variable.

default-values is also of a string type, providing default values for any file fields not provided in the file name, including default file options.

file-status is an integer variable that is primarily used to return a special status code if the file cannot be opened; this code allows a program to recover from an otherwise fatal error. The fourth parameter also may be used to determine the number of blocks allocated to a file or to specify the number of blocks to allocate to a new file. These uses are explained in detail below.

Commas must separate any optional parameters used; a comma must be included to mark off an omitted parameter but need not follow the last included parameter. The following example opens a file for direct access and skips the file-name parameter, indicating a temporary file.

```
rewrite(F1, , '/seek');
```

See "Random Access to Data Files" for details on the **/seek** switch, and see "I/O Control Switches" in the Programmer Reference for details on the use of other file switches.

The optional parameters may be used to redirect the standard files **input** or **output**, which by default are file variables associated with the standard terminal input or output devices, respectively. The next example redirects output from the terminal to the line printer.

```
rewrite(output, 'LP:');
```

Normally, an I/O error with **reset** or **rewrite** causes the support library to trap the error, terminate the program, and print an error message and procedure walkback. The fourth parameter may be used to return control to the program. If the fourth parameter is specified and an I/O error occurs, the support library sets the value of the fourth parameter to -1 and returns control to the program.

You must check the value returned by the fourth parameter and specify what action to take if an error occurs.

Example: Checking the Reset status parameter

```
reset(infile, Filename, '.lst', status); ———— default extension of .LST
if status = -1 then UserProcessError ———— response needed to error status
else ContinueUserProgram;
```

Or you may use the predefined functions in the support library to initiate run-time diagnostics. These functions check the status of the fourth parameter and respond accordingly. See "Run-Time Error Reporting" in the Programmer Reference. Either way, you must check the value of the fourth parameter each time you use it; otherwise, the program continues but may act unpredictably.

If the file is successfully opened, the fourth parameter returns the number of blocks allocated to the file. In addition, the fourth parameter may be used with **rewrite** to specify the number of blocks to be initially allocated to the file. When the size of the file is known in advance, this specification allows efficient space allocation by the operating system. If the file does not actually occupy the number of blocks specified, however, the operating system will truncate the file to the number of blocks needed. In turn, the value of the fourth parameter may be checked after a **rewrite** to be certain that the file was allocated the number of blocks you wished. If the fourth parameter is absent, the file size is determined by the operating system and expands dynamically. Examples:

Example: Checking the Rewrite Size parameter

```
reset(f,'test', '.pas', size);———————— assumes default of .PAS
writeln(size);-———————— returns the size of the file in blocks
:
:

size := 64;
rewrite(output, outstr, '.lis', size);———— file initially allocated 64 blocks
```

'Close' Procedure

The **close** predefined procedure indicates that its file parameter is no longer in use; **close** will reclaim buffer memory. Further access to the file is prohibited until **reset** or **rewrite** is used. Files are automatically closed upon program termination, or when they appear in another **reset** or **rewrite**; **close** allows files to be closed manually when it is necessary to reclaim buffer space before then. In addition, a file variable local to a procedure or function is automatically closed when that function or procedure terminates. See the sample program **Alphas** in the next section for implicit uses of **close**.

Random Access to Data Files ('Seek')

Pascal–2 includes the **seek** predefined procedure to allow direct access (random access) to data files opened with the **/seek** file control switch. The **seek** procedure requires two parameters: a file variable of the file to be accessed, declared as a **file of char** or other **file** type (but not of type **text**); and an integer record number (records in the file are numbered sequentially beginning with 1). After the **seek** call, the specified record is available in the file buffer if it exists; otherwise **eof** is set to indicate that the record is not available.

Seek also enables both reading and writing on the same file for in-place record updates. **Put** is required if the file buffer variable is to be written to the file. **Get** and **put** may be mixed with **seek** for sequential access, because the internal record pointer is updated after each **get** and **put**. See the example following for the use of **put** and **seek**.

After the file pointer is positioned by **seek**, both **read** and **write** as well as **get** and **put** may be performed. **Read** and **write** transfer data between the user variable and the file; **get** and **put** transfer data between the file buffer variable and the file. The following sequences may be used for direct access.

```
seek(F,I); read(F,V);  { read record I into V  }
seek(F,I); write(F,V); { write record I from V }
```

Example: Direct (Random) Access Using 'Seek'

```
program Alphas;
  var
    C: char;
    F: file of char;

  begin
    rewrite(F, 'alpha.txt'); ————————— open F for writing
    for C := 'a' to 'z' do
      write(F, C); ————————— write letters of the alphabet to F
    reset(F, 'alpha.txt/seek'); ————————— close and reopen F for seeking
    seek(F, 4); ————————— read record containing 'd'
    writeln(F^); ————————— write a 'd' to output
    F^ := 'z'; ————————— 'd' becomes 'z'
    put(F); ————————— write 'z' to F in place of 'd'
  end. ————————— F closed automatically
```

As the program shows, the /seek file control switch must be used with reset or rewrite if the seek procedure is to be used to access the file. See "I/O Control Switches" in the Programmer Reference for details.

At run-time, the character 'd' is written to the terminal. After program termination, the file ALPHA.TXT contains:

`abczefghijklmnopqrstuvwxyz` ———— z takes the place of d

Seek does not work on text ~~files.~~ For simulated random access on text files, you must use the getpos and setpos external procedures. See "Random Access to 'Text' Files" in the Programmer Reference.

String Input ('Read' and 'Readln')

A character string is a packed array [1..n] of char. The read and readln procedures may be used to read variables of string types. Characters are read until the variable is filled. If eoln becomes true, the remainder of the string is filled with spaces. See "The Dynamic String Package" in the Utilities Guide for more sophisticated ways to read and manipulate strings.

'Break' Procedure

For efficiency, Pascal–2 buffers transmitted output data. Break(F) forces the actual transmission of data from a partially filled buffer of file F. This can be used with interactive terminals to print the terminal buffer to the screen. The cursor is positioned at the end of the line.

Octal Output

In an integer write procedure call, a negative field-width specification will represent characters in octal (base 8).

Example:

```
write(I:-5);                 { Display octal value of I }
```

Real Number Formatting

If the second formatting field is negative, a real number is printed in scientific notation. The number of digits to the right of the decimal point is the number specified in the second field. (The standard allows you to specify an integer constant or an integer expression in either formatting field.)

For example,

```
write(R:20:-5);
```

prints R with one digit to the left of the decimal point and five digits to the right, followed by an upper-case E, a sign character '+' or '-' and two digits signifying the exponent. The entire number is right-justified in a 20-character field.

If R has the value −367.2, the statement writeln('R=',R:20:-5) prints:

```
R=          -3.67200E+02
```

Low-Level Interface

This section describes Pascal–2 extensions that are useful to programmers needing access to machine-dependent characteristics.

Boolean Operators on Integer

The boolean operators **and**, **or**, and **not** may be applied to operands of `integer` or `integer` subrange type. The **not** operator is always applied first. The operators produce a 16-bit result of `integer` type.

Nondecimal Integer Constants

Nondecimal integer constants may be specified in two forms of notation. In the preferred form, the nondecimal value is written as shown:

nondecimal-integer-constant = *digit-sequence* "#" *hexadecimal-digit-sequence* .

where *digit-sequence* is the radix, or base, of the number, in the range 2..16. The number following the cross-hatch character '#' is any number represented in base *digit-sequence* notation. The '#' symbol is required regardless of base. For example, the decimal value 255 is written `8#377` for base 8 and `16#FF` for base 16. Also, the redundant form `10#255` is valid for the decimal value 255.

Pascal–2 supports another form of notation as a special case. Octal (base 8) notation for integer constants is signified by the suffix "`B`" (upper or lower case), so that `377B` and `377b` are the same value as 255 decimal.

Extended-Range Arithmetic

The normal range of `integer` variables in Pascal–2 is −32767..32767, but you may also declare `integer` types in the extended range of 0..65535. A variable with an upper limit greater than 32767 is called an extended-range or "unsigned" variable. Any integer value may be assigned to an unsigned variable and is converted to an unsigned value according to the number's underlying bit pattern, with the "sign" bit of the signed integer being considered a "significant" bit of the extended-range integer. If the integer value being assigned is negative, no error is trapped at run-time, since there is no way for the compiler to tell the difference between a negative "signed" value and an extended-range "unsigned" value. The same sort of implicit transformation is true when an extended value is assigned to an integer variable.

Addition, subtraction, and multiplication are signed operations and are performed on extended-range variables in the same way that they are performed on signed variables. Division and modulo are unsigned operations only for dividends in the extended range; they do not treat divisors greater than 32767 as unsigned values. Comparison operations are signed or unsigned according to variable type.

The following sample program illustrates the way Pascal–2 handles extended-range numbers. Within the `repeat until` statement, the program reads an integer then prints it as an unsigned integer and as a signed integer. The external procedure **Uwrite** is provided in the section on "Unsigned Integer Conversion" in the Programmer Reference.

Example: Signed/Unsigned Integer Conversion Program

```
program BigNumberTest;
  type
    Unsigned = 0..65535;

  var
    BigNumber: Unsigned;

  procedure Uwrite(I: Unsigned; Width: integer);
    external;     { procedure to write an unsigned integer to output }

begin    { BigNumberTest }
  repeat
    write('Enter an integer: ');
    readln(BigNumber);
    write('  Unsigned, BIGNUMBER = ');
    uwrite(BigNumber,1);  writeln;
    writeln('  Signed, BIGNUMBER   = ', BigNumber:1);
    writeln;
  until false { forever };
end.     { BigNumberTest }
```

The program is executed, producing the following results. As mentioned earlier, the allowable range of values for the integer **BigNumber** is −32767..32767. The final entry — in fact, any value outside the range of possible integers — is an invalid value for an integer, halting the program with a walkback (unless **walkback** is disabled).

```
Enter an integer: -1
  Unsigned, BIGNUMBER = 65535
  Signed, BIGNUMBER   = -1

Enter an integer: -32767
  Unsigned, BIGNUMBER = 32769
  Signed, BIGNUMBER   = -32767

Enter an integer: 32767
  Unsigned, BIGNUMBER = 32767
  Signed, BIGNUMBER   = 32767

Enter an integer: -5555
  Unsigned, BIGNUMBER = 59981
  Signed, BIGNUMBER   = -5555

Enter an integer: 5555
  Unsigned, BIGNUMBER = 5555
  Signed, BIGNUMBER   = 5555

Enter an integer: 65535
```

```
TT2 -- I/O error at user PC= 1332
Illegal value for integer
I/O error code= 19. (23B) in file: TI:

Error occurred at line 16 in program bignumbertest
```

"Origin" Declaration

A variable can be declared to have a particular address in the I/O page or system area with the following syntax:

var-declaration = *var-element* {"," *var-element*}":" *type* .

var-element = *identifier* ["origin" *constant*] .

The constant in the above syntax must have an integer value. A variable so specified has the address given by the integer following origin. This must be in the system space 0..777B or in the I/O page 160000B..177777B.

The following example demonstrates the use of origin, plus the use of the ref and size functions. See "Ref Function" and "Size and Bitsize Functions" for more details on those routines. The example controls a mythical device. The procedure ReadData sets up the device's control registers and initiates a transfer from the device into the task's memory. This example is specific to a machine without memory management hardware, such as a small RT-11 system.

Example: Use of 'Origin' 'Ref', 'Size' and 'Bit Size'

```
program Device;                      { example of device control }

const
  Ready = 200B;                      { ready flag }
  ReadBuffer = 1;                    { read data command }

type
  Buffer = packed array [1..100] of char;
  BufferPointer = ^Buffer;

var
  StatusRegister  origin 177316B: integer;
  ControlRegister origin 177314B: integer;
  BufferAddress   origin 177312B: Buffer_Pointer;
  ByteCount       origin 177310B: integer;
  Data: Buffer;                      { holds data from device }

  procedure ReadData;
    begin    { ReadData }
      BufferAddress := ref(Data);          { Address for DMA xfer }
      ByteCount := size(Buffer);           { size of buffer }
      ControlRegister := ReadBuffer;       { start transfer }
      { Wait for device to complete transfer }
      while (StatusRegister and Ready) = 0 do {wait};
    end;    { ReadData }

begin    { Device }
  ReadData;
end.     { Device }
```

'Ref' Function

The **ref** function, with a variable argument of type **T**, produces a pointer to that variable with result type **^T** (pointer to **T**). The **dispose** routine cannot always detect attempts to dispose of a pointer generated with this function, and you should not try to do so.

See the example under "Origin Declaration" and under "Loophole Function" for uses of **ref**.

'Size' and 'Bitsize' Functions

Two functions, `size` and `bitsize`, give the programmer information on the space allocated for values of different types. The functions have a single argument, a type identifier.

The function `size` returns the number of bytes that would be allocated for an object of that type by normal variable allocation. The function `bitsize` returns the number of bits that would be allocated for an object of that type as a component of a packed record. This is the actual number of bits required to hold the value.

For example, suppose you had declared a type `Subrange = 0..15` and called the functions `size` and `bitsize`, as in the following example program. The results tell you that two bytes and four bits are allocated for the argument in question.

```
program SizeBitsize;
type
  Subrange = 0..15;

begin
  writeln(size(Subrange));
  writeln(bitsize(Subrange));
end.
```

The program yields these results:

```
2  ————————————— 2 bytes are allocated
4  ————————————— 4 bits are allocated
```

These functions are primarily useful when you are interfacing with the operating system or with hardware functions.

See "Origin Declaration" for another example of `size`.

'Loophole' Function

The invocation of `loophole` requires two parameters:

`loophole(`*returned-type*`,` *expression-to-convert*`);`

where *returned-type* is an identifier specifying the data type to be returned by `loophole`, and *expression-to-convert* is an expression of a "compatible" type that is converted to *returned-type*. In this context two types are considered compatible only if they require the same amount of storage (see "Storage Allocation" in the Programmer Reference), or if they are both non-real scalar types.

The result of the `loophole` function is the bit pattern of the second argument, expressed as a value of the type specified in the first argument.

The following program illustrates the compatibility rules that govern the use of `loophole`. The program coerces a `real` number to an equivalent two-word array of integers representing the two words used to store the real value, then coerces the two-word array back into a real number. The program then coerces an integer in the range 0..4 to a scalar of type Car, then coerces the scalar back to an integer. The loophole(integer,S) is equivalent to the statement I := ord(S).

Example: Type Coercion Integer to Scalar

```
program Coerce;

  type
    Realequiv = array [0..1] of integer;
    Car = (Buick, VW, Datsun, Chevy, BMW); { scalar type }

  var
    Re: Realequiv;
    R: real;
    S: Car;    { scalar }
    I: integer;

  begin   { Coerce }
    write('Enter a Real number: ');
    readln(R);
    Re := loophole(Realequiv, R); { coerces real into 2-wd array of integers }
    writeln('Re = ', Re[0]:-8, Re[1]:-8);  { 2-wd array printed in octal }
    R := loophole(Real, Re);  { coerces 2-wd array back to real }
    writeln('R  = ', R);
    write('Enter an integer in range 0..4: ');
    readln(I);
    S := loophole(Car, I);    { coerces integer to scalar }
    write('S  = ');
    case S of      { writes the scalar value }
      Buick: writeln('Buick');
      VW: writeln('VW');
      Datsun: writeln('Datsun');
      Chevy: writeln('Chevy');
      BMW: writeln('BMW');
      end;  { case }
    I := loophole(integer, S); { coerces scalar back to integer }
    writeln('I  =', I);
  end.   { Coerce }
```

When executed, the program yields these results:

```
Enter a Real number: 21567.9
Re =     43650    77715 ———————— octal representation of real number (2 words)
R  =  2.156790E+04
Enter an integer in range 0..4: 2
S  = Datsun
I  =       2
```

The only other method of type coercion is to declare a record with variants, using the fact that the compiler overlays storage for different variants. This method makes the same kind of assumptions as the loophole function about the compiler's allocation of memory and machine's architecture. However, the **loophole** function has several advantages over variant records:

- No assumption need be made about field allocation in a variant record.
- The compiler checks that the different types are the same size.
- The bypassing of type checking rules is clearly marked (the compiler flags **loophole** if the **$standard** switch is set). Also, if the code is used with a compiler other than Pascal-2, that compiler should mark **loophole** as an error, and appropriate changes can be made to the code. With variant records, the code might compile but not work.

The following sample program uses the **loophole** function to perform arithmetic on pointers so that a block of the task's memory can be printed.

```
program MDump;

  type
    Word = 0..65535;

    procedure DumpMemory(Start, Finish: Word);
      type
        Pointer = ^integer;
      var
        P: Pointer;

      begin     { Dump Memory }
        P := loophole(Pointer, Start);
        while loophole(Word, P) <= Finish do begin
          writeln(loophole(integer, P): -6, ': ', P^: -6);
          P := loophole(Pointer, loophole(Word, P) + 2);
          end;
      end;      { Dump Memory }

  begin  { MDump }
    DumpMemory(1210B, 1220B);
  end.   { MDump }
```

The program yields these results:

```
1210:      6
1212: 101032
1214:  10546
1216:  12746
1220: 177772
```

Non-Standard Language Elements

Language features described in this section are deviations from standard Pascal.

Program Parameters

According to the standard, parameters supplied in the program header indicate external files. Further, the **input** and **output** files must appear in the program header if they are used in the program. The **input** and **output** files are always defined at the global level and may not be redeclared at that level.

With Pascal–2, the program header is not required, and any program parameters are entirely ignored (see "Program Heading" under "Syntax Extensions"). External files are referenced instead by an extended form of **reset** and **rewrite** using a second parameter (a string) giving the external filename (see "External File Access" under "I/O Support Extensions").

Directives

The standard treats directives such as **forward** as neither an identifier nor a reserved word. Pascal–2 treats the directives **forward**, **external**, **nonpascal** as reserved words. An identifier cannot have the same name as one of these directives.

'Mod' of Negative Numbers

The standard states that the divisor must be positive and the operator **mod** must have a non-negative result. That is,

$$0 \leq I \textbf{ mod } J < J$$

. The Pascal–2 compiler generates a divide instruction that gives a negative result if I is negative. The standard result is generated by:

```
Result := I mod J;
if Result < 0 then Result := Result + J;
```

Returning of Structured Types

Under the standard, functions return simple data types only (e.g., **integer**, **real**, **char**). With Pascal–2, functions may return structured data types such as record, **array** and **set** types in addition to simple types. For example:

```
function KeySort(Key: KeyType): StructType;
```

where *StructType* is the structured data type of the return value of **KeySort**.

Additional Predefined Functions and Procedures

The Pascal-2 system includes predefined functions and procedures, as allowed by the standard. Most of these are grouped according to function in other subsections in this specification. This subsection describes miscellaneous predefined functions and procedures not otherwise described.

Because these procedures and functions are known to the compiler, they need not be declared in the program. The only exception is `timestamp`, which is functionally similar to the other procedures and functions but is not yet predefined. `Timestamp` is not predefined so it must be declared as an external procedure.

.Procedure 'Delete'

The predefined `delete` procedure allows the deletion of a single file that is opened in a Pascal program. `Delete` accepts one argument, the file variable of the file to be deleted. Invoke the procedure with a statement similar to the following:

```
delete(F);
```

Internally, this procedure closes and deletes the file specified by the argument. Your program should not close the file (using `close`) before invoking the `delete` procedure. The run-time error message "can't delete file" results if the file cannot be deleted for some reason. See the example after the discussion of the `rename` procedure.

Procedure 'Rename'

The predefined procedure `rename` allows the renaming of an open file, from within a Pascal program. `Rename` accepts two arguments. The first argument passed to `rename` must be the file variable of the original file name. The second argument must be the file name of the new file. Invoke the procedure with a statement similar to the following.

```
rename(F, 'newfil.txt');———— renames F to NEWFIL.TXT
      or:
NewF := 'newfil.txt;8';
rename(F, NewF);———— renames F to NEWFIL.TXT;8
```

The second argument may be a constant, a variable, or a literal string. A version number may be specified. If the version number of the new file is omitted, `rename` makes the new file the current version. The second argument must contain at least one field. If any fields are omitted from the second argument, the omitted field takes the corresponding value from the original file name. For example, to change the extension only, use a statement similar to this:

```
rename(F, '.mac');-- file name is the same; .MAC is the new extension
```

The original file must be open (via `reset`) before `rename` may be called on the file. The renamed file is automatically closed upon completion of the operation.

The next program illustrates the use of the delete and rename predefined procedures. The program reads a file of weather observations and weeds out duplicate reports, or "dupes." The "good" reports are written to a file, which is later renamed. The file of duplicate reports is then deleted.

Example: Use of Predefined Procedures 'Delete' and 'Rename'

```
program Dupes;

  const
    Climat_File = 'climat.dat';

  var
    Data_File: text;        { file of weather observations }
    Dupe_File: text;        { file of duplicate reports }
    Good_File: text;        { file of good reports minus duplicates }

  procedure Discard_Dupes(var F, G, H: text);
    external;
      { This procedure sorts F, a file of weather observations,
        saving good reports on file G and discarding duplicate
        reports on file H. }

  begin      { Dupes }
    reset(Data_File, 'weax.dat');
    rewrite(Dupe_File, 'dupe.tmp');
    rewrite(Good_File, 'good.dat');
    Discard_Dupes(Data_File, Good_File, Dupe_File); { Weed out the dupes }
    rename(Good_File,Climat_File);-------- renames GOOD.DAT to CLIMAT.DAT
    delete(Dupe_File);------------------- deletes the file of duplicates
  end.       { Dupes }
```

Predefined Function 'Time'

The predefined function time takes no parameters and returns a real value corresponding to the current time of day. The value time is represented in hours after midnight, so that 9:30 a.m. is 9.50 and 1:45 p.m. is 13.75. The resolution of time depends on the operating system, but all operating systems provide a resolution of at least one second.

The value returned could be used in header information. (If you wanted the date as well as the time, you would use timestamp, described later.) Or you could call time at the beginning and end of a text-processing program and write a procedure that calculates the number of lines processed per minute, based on the difference in value returned. Or, because it generates a real number, time may be used to "seed" a pseudo-random number generator. The next example returns uses time to return the time of day. Chr(7) is the "bell" character.

Example: Use of TimeStamp

```
program WriteTime;

  var
    Hrs, Mins: integer;
    AmPm: packed array[1..2] of char;

  Mins := Round(time * 60);
  Hrs := Mins div 60;
  Mins := Mins mod 60;
  if (Hrs < 12) then AmPm := 'AM'
  else if (Hrs = 12) and (Mins = 0)
    then AmPm := 'M ' else AmPm := 'PM';
  write('At the tone the time will be: ');
  write(((Hrs+11) mod 12 + 1):2);
  write(':', Mins div 10:1, Mins mod 10:1, AmPm:3);
  writeln(Chr(7));
end.       { WriteTime }
```

Running the program prints these results:

```
At the tone the time will be: 11:37 AM < beep>
```

Procedure 'TimeStamp'

The timestamp procedure provides a way to obtain the date and time from within a Pascal program. Date and time are obtained simultaneously so that they are consistent, even close to midnight.

Timestamp is included in the Pascal-2 library, but the name is not pre-declared by the compiler. You must include a definition similar to:

```
procedure Timestamp(var day, month, year,               { date }
                    hour, min, sec: integer );          { time }
  external;
```

The next program prints the date and time using timestamp.

```
program DateTime(output);
  var
    Day, Month, Year: Integer;      { date data }
    Hour, Minute, Second: Integer;  { time data }

  procedure Timestamp(var Day, Month, Year,                    { date }
                          Hour, Min, Sec: Integer );           { time }
    external;

  procedure PrintTwo(N: Integer);
    begin { Print a number on the output file with two digits, including
            leading zeros if needed. The number must be 99 or less }
      write(output, N div 10: 1, N mod 10: 1);
    end; { PrintTwo }

  begin     { DateTime }
    Timestamp(Day, Month, Year, Hour, Minute, Second);
    PrintTwo(Day);
    case Month of
       1: write(output, '-Jan-');
       2: write(output, '-Feb-');
       3: write(output, '-Mar-');
       4: write(output, '-Apr-');
       5: write(output, '-May-');
       6: write(output, '-Jun-');
       7: write(output, '-Jul-');
       8: write(output, '-Aug-');
       9: write(output, '-Sep-');
      10: write(output, '-Oct-');
      11: write(output, '-Nov-');
      12: write(output, '-Dec-');
      end;
    write(output, Year: 4, ' ');
    PrintTwo(Hour);
    write(output, ':');
    PrintTwo(Minute);
    write(output, ':');
    PrintTwo(Second);
    writeln(output);
  end.      { DateTime }
```

The results of the program are:

```
16-Jun-1983 14:28:31
```

Error Handling

This section describes the errors defined by the Pascal standard and Pascal–2's handling of them.

Detected Errors

Pascal–2 detects the following errors in all cases:

1. `Ln` or `sqrt` has a negative argument.
2. The integer value returned by `trunc` or `round` lies outside the range `-maxint..maxint`.
3. Integer or real division by zero.
4. The result of a real operation cannot be expressed because of limitations in the floating-point format.
5. No label matches the value of the case index in a `case` statement.
6. The characters being read from a text file do not represent a legal value for the type of variable being read.
7. An attempt to call `get`, `read`, or `readln` when the file has not been `reset` or when `eof` is `true` for that file.
8. An attempt to call `put`, `write`, `writeln`, or `page` when the file has not been rewritten or when `eof` is `false` for that file.
9. A call to `put` when the file variable is undefined.

Pascal–2 detects the following errors under these conditions:

1. The value assigned to a variable or value parameter is not within the declared range of values for that variable. Detected when the `$rangecheck` compiler switchis enabled. (Default.) Not detected when a negative value is assigned to an extended-range variable. See "Extended-Range Arithmetic" for more details.
2. An index expression for an array access is outside the range of the corresponding index type. Detected when the `$indexcheck` switchis enabled. (Default.)
3. A reference through a pointer with a `nil` or undefined value. Reference through a `nil` pointer is detected when the `$pointercheck` switchis enabled. (Default.) Reference through an undefined value is not detected, although many cases will be detected at compile time.
4. In a `for` statement, the initial and final values are not within the range of the controlled variable when the initial value is assigned to the controlled variable. Detected when the $rangecheck switchis enabled. (Default.)
5. The calling of `dispose` with a `nil` or undefined parameter. Detected if the parameter is `nil`; detected if the parameter was made

undefined by a previous **dispose**. The **dispose** of an undefined pointer is sometimes detected.

6. The result of the **sqr** function is out of range. Detected if the argument type is **real**; undetected if the argument type is **integer**.
7. The result of **chr(x)** is not within the character set. Detected only if a value is assigned to a variable or is passed as a parameter.
8. The result of **succ** or **pred** lies outside the range of the type. Detected only if the value then is assigned to a variable or is passed as a parameter.
9. A **mod** with the right-hand side less than or equal to zero. Detected if the value is zero; otherwise not.
10. Reference to an undefined variable. Undetected in general. However, many simple cases are detected at compile time.
11. A return from a function without a value being assigned to the function. Undetected in general. However, many simple cases are detected at compile time.
12. An attempt to call **put** on a file that was opened with **reset**. Detected except for a file with the **/seek** file control switch specified when the file was opened.

Undetected Errors

Pascal–2 does not detect the following errors:

1. A set value assigned to a set variable or value parameter contains members not in the range of the base type of the set variable.
2. An access to a field in a variant record that is not selected by the current value of the tag-field.
3. A **dispose** of a variable allocated on the heap while there is an active reference to that variable as a variable parameter or in a **with** statement.
4. A change in the value of a file variable by a **get** or **put** while there is an active reference to that variable as a variable parameter or in a **with** statement.
5. Accessing of a variable allocated with **new**($p, c_1, \ldots, c_n$) as an entire variable, in an assignment or as a parameter.
6. Calling of **dispose(p)** when the value of **p^** was created with **new**($p, c_1, \ldots, c_n$), or calling of **dispose**($p, c_1, \ldots, c_n$) with a variable created with **new** and a different set of tag values.
7. The result of an integer operation is incorrect because of overflow.
8. The value of a format expression to a **write** statement is less than 1. Undetected (used in a language extension).

Appendix A: Predefined Identifiers

Constants

- False
- Maxint
- True

Types

- Boolean
- Char
- Integer
- Real
- Text

Variables

- Input
- Output

Functions

- Abs
- Arctan
- Bitsize*
- Chr
- Cos
- Eof
- Eoln
- Exp
- Ioerror*
- Iostatus*
- Ln
- Loophole
- Odd
- Pred
- Ref*
- Round
- Sin
- Size*
- Sqr
- Sqrt
- Succ
- Time*
- Trunc

Procedures

- Break*
- Close*
- Delete*
- Dispose
- Get
- New
- Noierror*
- Pack
- Page
- Put
- Read
- Readln
- Rename*
- Reset
- Rewrite
- Seek*
- Unpack
- Write
- Writeln

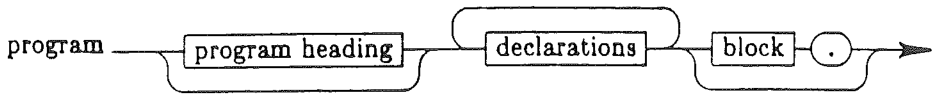
Appendix B: Reserved Words

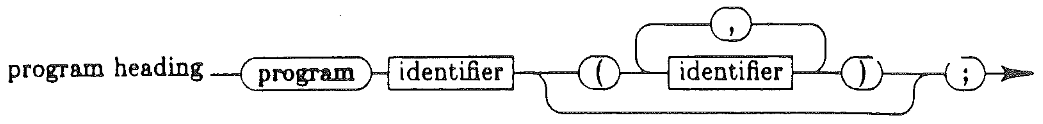
- And
- Array
- Begin
- Case
- Const
- Div
- Do
- Downto
- Else
- End
- External*
- File
- For
- Forward
- Function
- Goto
- If
- In
- Label
- Mod
- Nil
- Nonpascal*
- Not
- Of
- Or
- Orgin*
- Otherwise*
- Packed
- Procedure
- Program
- Record
- Repeat
- Set
- Then
- To
- Type
- Until
- Var
- With
- While

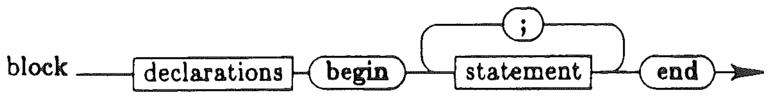
*Items marked with the asterisk are extensions of standard Pascal.

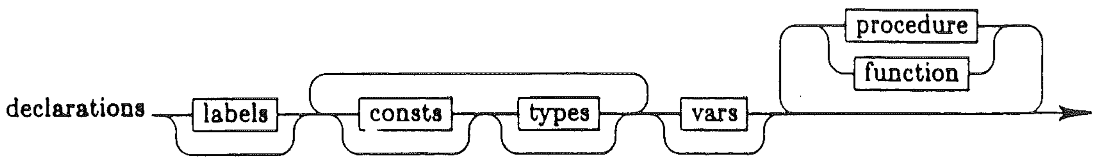
Appendix C: Pascal–2 Syntax

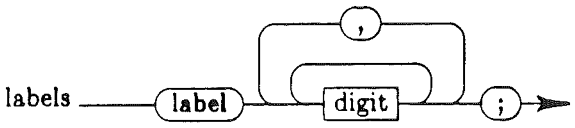
Pascal–2 Syntax Diagrams

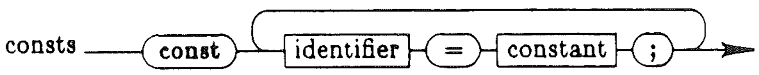


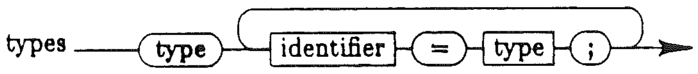


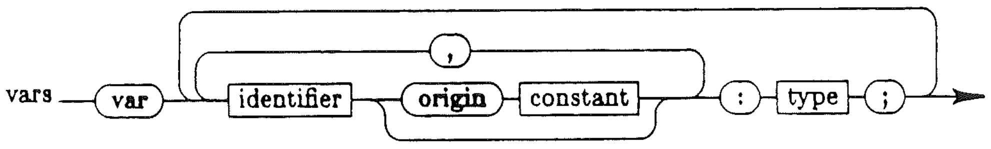


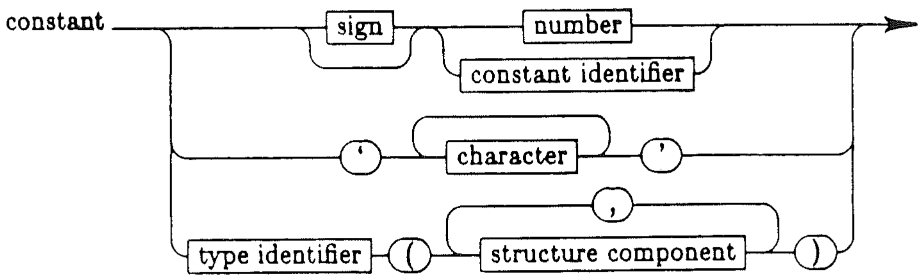


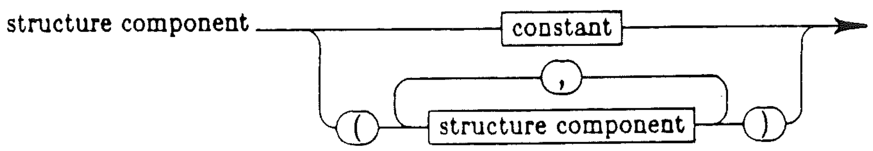


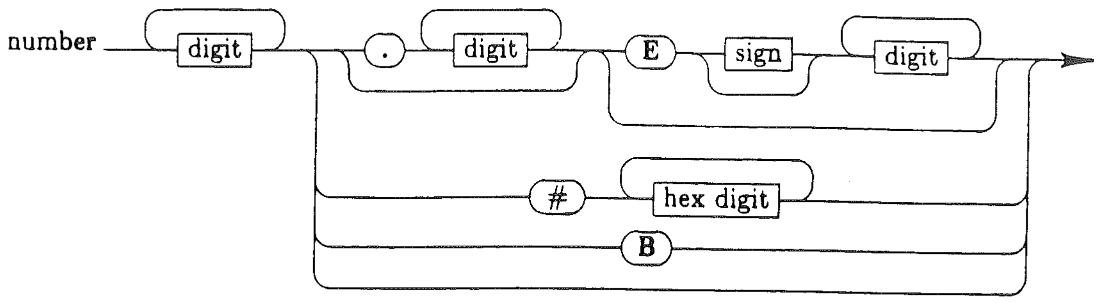


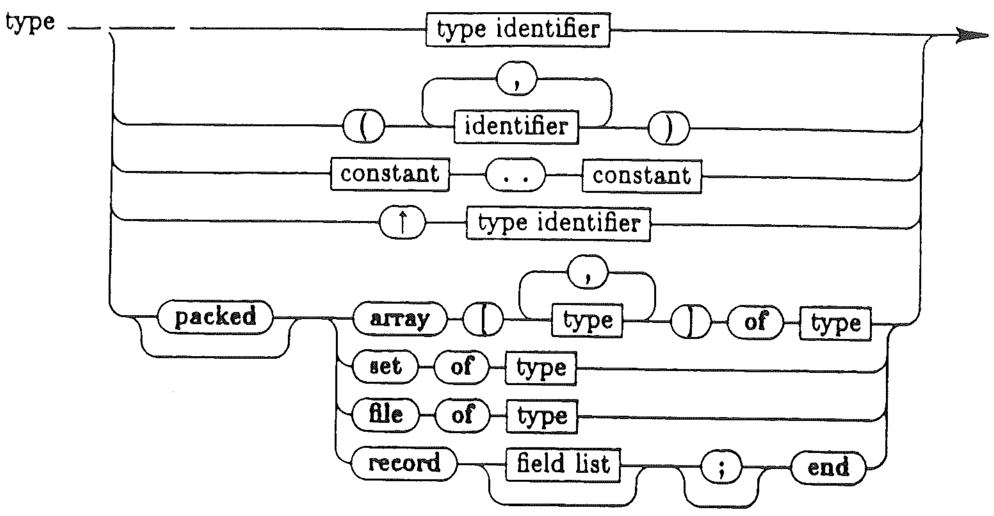


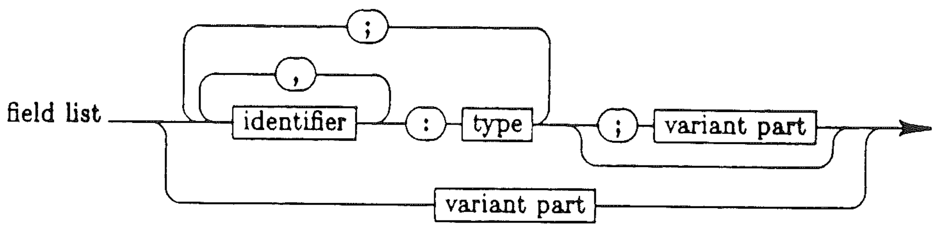


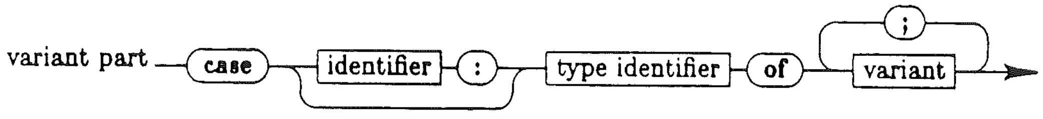


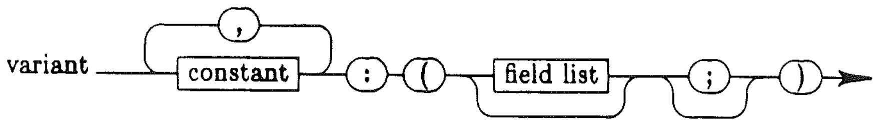


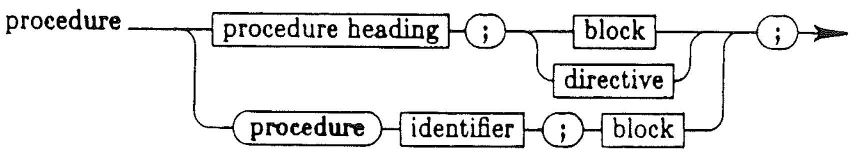

number
digit
.
digit
E
sign
digit
#
hex digit
B

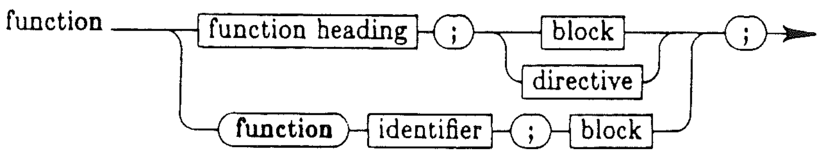

type
type identifier
(
,
identifier
)
constant
. .
constant
↑
type identifier
packed
array
[
,
type
]
of
type
set
of
type
file
of
type
record
field list
;
end

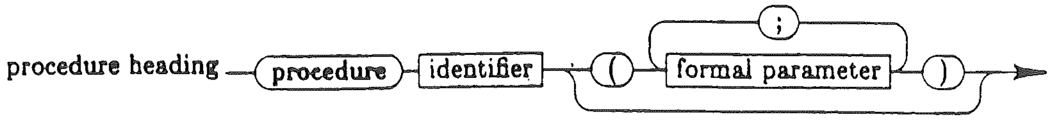

field list
;
,
identifier
:
type
;
variant part
variant part

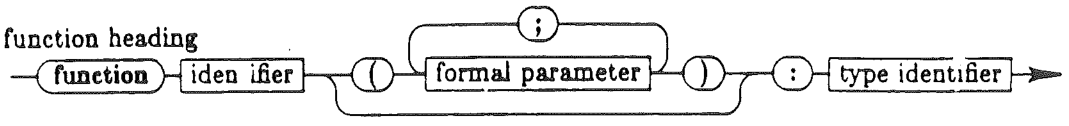

variant part
case
identifier
:
type identifier
of
;
variant

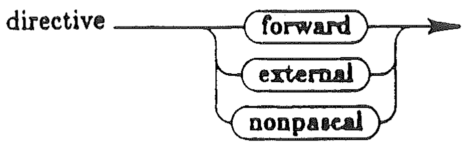

variant
,
constant
:
(
field list
;
)

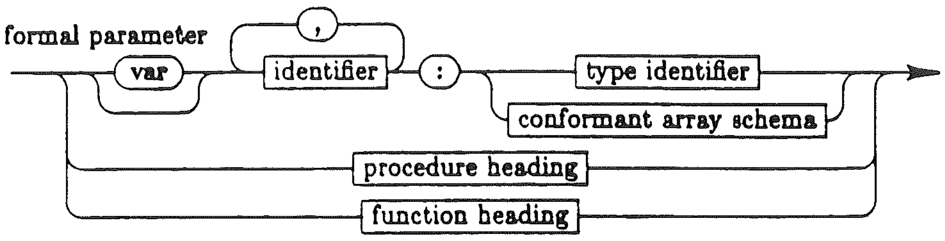

procedure
procedure heading
;
block
directive
;
procedure
identifier
;
block

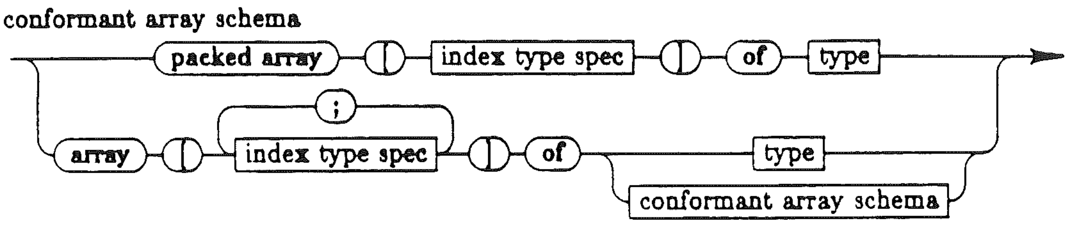

function
function heading
;
block
directive
;
function
identifier
;
block

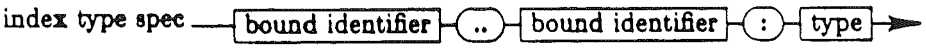

procedure heading
procedure
identifier
(
formal parameter
;
)

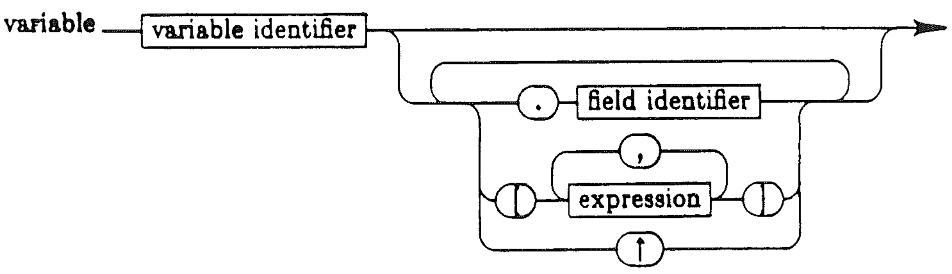

function heading
function
iden ifier
(
formal parameter
;
)
:
type identifier

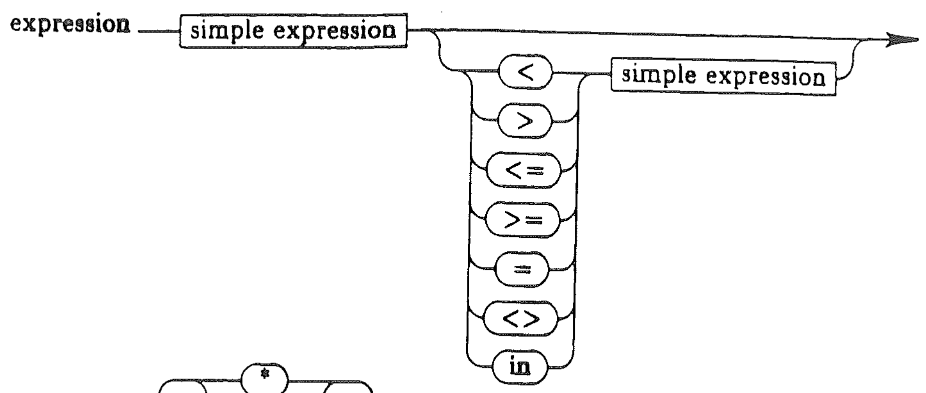

directive
forward
external
nonpascal

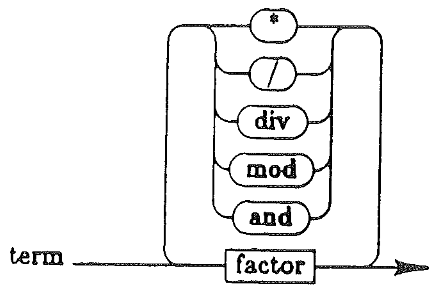

formal parameter
var
identifier
,
:
type identifier
conformant array schema
procedure heading
function heading

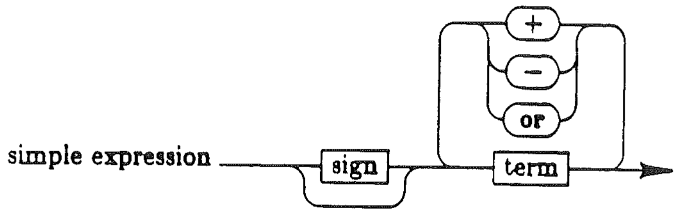

conformant array schema
packed array
[
index type spec
]
of
type
array
[
index type spec
;
]
of
type
conformant array schema

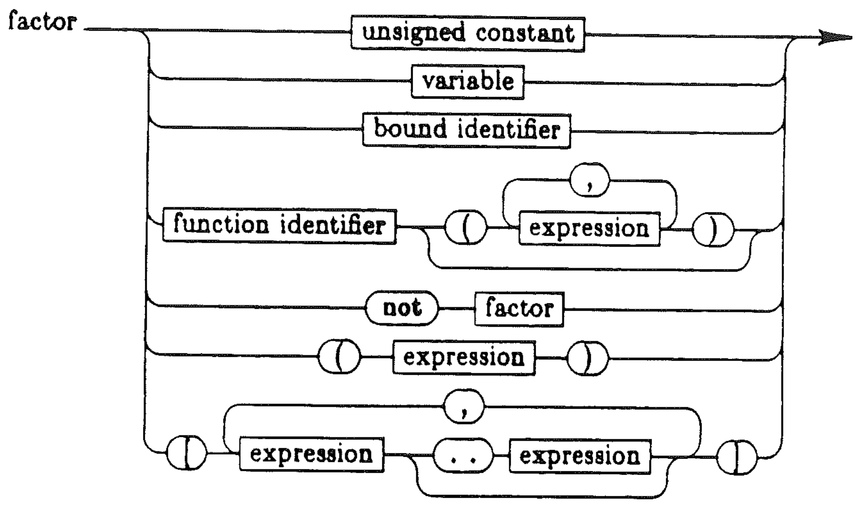

index type spec
bound identifier
..
bound identifier
:
type

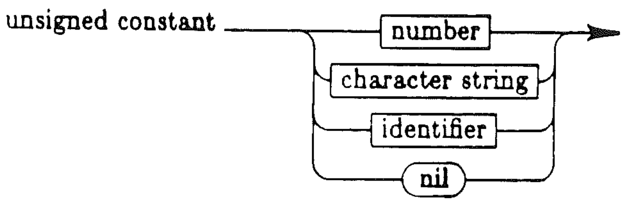

variable
variable identifier
.
field identifier
[
expression
,
]
↑


expression
simple expression
<
>
<=
>=
=
<>
in
simple expression

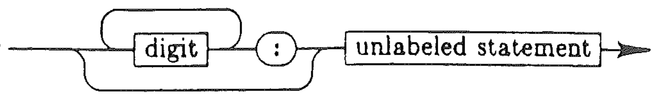

*
/
div
mod
and
term
factor


+
–
or
simple expression
sign
term

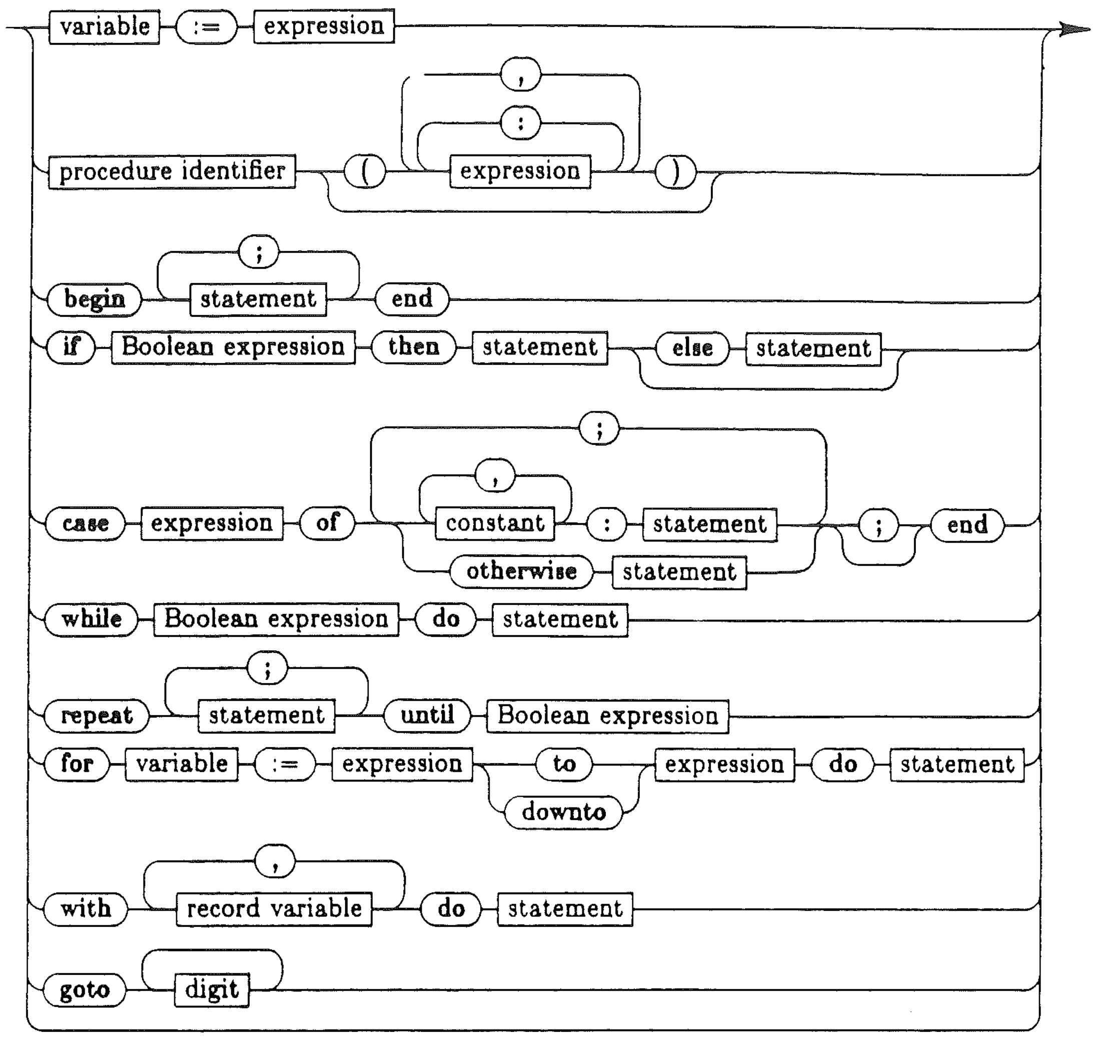

factor
unsigned constant
variable
bound identifier
,
function identifier
(
expression
)
not
factor
(
expression
)
,
[
expression
..
expression
]


unsigned constant
number
character string
identifier
nil

statement



unlabeled statement



Extended Backus-Naur Form

The notation used for describing syntax in this guide is a variant of the Backus-Naur Form (BNF) originally developed to describe the syntax of Algol 60. This particular variant was proposed by Niklaus Wirth ("What Can We Do About the Unnecessary Divergence of Notations for Syntactic Definitions?", *Communications of the ACM,* November 1977, vol. 20, number 11).

A "terminal symbol" is a symbol that actually appears in the language itself. Examples of terminal symbols in Pascal are:

`begin   +   >=`

Terminal symbols are written in quotes, e.g.: "`terminal`".

Some terminal symbols are not easily expressed in this way, and these may be represented by comments contained in angle brackets <>. For example:

<*any printable character*>

A "nonterminal symbol" is used in the description of the language but does not actually appear in the text of the language. That is, it is used to talk about the language. A nonterminal symbol stands for some sequence of terminal or nonterminal symbols. Nonterminal symbols are written without quotes. For example:

identifier, interface-part

A "production" is a rule specifying which terminal and nonterminal symbols make up another nonterminal symbol. A production is written:

left-hand-side = *right-hand-side* .

The *left-hand-side* is a nonterminal symbol; the *right-hand-side* is some combination of terminal and nonterminal symbols. A production indicates that the *left-hand-side* is made up of the symbols on the *right-hand-side.* A production is terminated with a period.

Within a right-hand-side, the following operators may occur:

(blank) indicates that the two symbols are concatenated. For example:

lhs = "a" "b" "c" .

indicates that *lhs* consists of the string abc.

| (vertical bar) indicates that the two symbols are alternatives. Concatenation is performed before alternation. For example:

lhs = "ab" | "cd" .

indicates that *lhs* consists of one of the strings ab, cd.

[ ] (brackets) indicate that the enclosed symbols are optional. For example:

lhs = "a" ["bc"] "d" .

indicates that *lhs* consists of one of the strings abcd, ad.

{ } (braces) indicate that the enclosed symbols are repeated zero or more times. For example:

lhs = "a" {"b"} "c" .

indicates that *lhs* consists of any of ac, abc, abbc, abbbc, ...

( ) (parentheses) are used for grouping as they are in mathematics.

We can now use this notation to describe itself as an example. The productions for *letter*, *digit* and *character* are not given here but are obvious.

syntax = { *production* } .

production = *nonterminal-symbol* "=" *expression* "." .

expression = *term* { "|" *term* } .

term = *factor* { *factor* } .

factor = *nonterminal-symbol* | *terminal-symbol* | "(" *expression* ")"
| "[" *expression* "]" | "{" *expression* "}" .

terminal-symbol = """ *character* { *character* } """ | <*any comment in angle brackets*> .

nonterminal-symbol = *letter* { *letter* | *digit* | "-" } .

Pascal-2 Lexical Description

This set of productions defines the lexical representation of Pascal-2. Productions that differ from the standard are marked with an asterisk (*). The case of any alphabetic character is insignificant except in a *character-string*. Lower-case is used in this description.

1.* *letter* = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "$" .

2. *digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

3.* *octal-digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .

4.* *hexadecimal-digit* = *digit* | "a" | "b" | "c" | "d" | "e" | "f" .

5. *special-symbol* = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" |
"(." | ".)"
| "." | "," | ":" | ";" | "^" | "@" | "(" | ")" |
"<>" | "<=" | ">="
| ":=" | ".." | *word-symbol* .

6.* *word-symbol* = "and" | "array" | "begin" | "case" | "const" |
"div" | "do"
| "downto" | "else" | "end" | "file" | "for" |
"function" | "goto" | "if"
| "in" | "label" | "mod" | "nil" | "not" | "of" |
"or" | "origin" | "otherwise"
| "packed" | "procedure" | "program" | "record"
| "repeat" | "set" | "then"
| "to" | "type" | "until" | "var" | "while" |
"with" .

7.* *identifier* = *letter* { *letter* | *digit* | "_" } .

8. *bound-identifier* = *identifier* .

9.* *directive* = "forward" | "external" | "nonpascal" .

10. *digit-sequence* = *digit* { *digit* } .

11.* *unsigned-integer* = [*digit-sequence* "#"] *hexadecimal-digit-sequence* .

12. *unsigned-real* = (*unsigned-integer* "." *digit-sequence* ["E" *scale-factor*])
| (*unsigned-integer* "E" *scale-factor*) .

13.* *nondecimal-integer* = *digit-sequence* "#"
(*hexadecimal-digit* { *hexadecimal-digit* } | *octal-number*) .

14.* *octal-number* = *octal-digit* { *octal-digit* } "b" .

15.* *unsigned-number* = *unsigned-integer* | *unsigned-real* | *octal-number* .

16. *scale-factor* = *signed-integer* .

17. *sign* = "+" | "-" .

18. *signed-integer* = [*sign*] *unsigned-integer*;

19. *signed-real* = [*sign*] *unsigned-real*;

20.* *signed-number* = *signed-integer* | *signed-real* | [*sign*] *octal-number* .

21. *label* = *unsigned-integer*;

22. *character-string* = "'" *string-element* { *string-element* } "'" .

23. *string-element* = "''" | <*any printable ASCII character*> .

24. *comment* = ("{" | "(*")
<*any sequence of characters and ends of lines not containing* "}" *or* "*)"*>
("}" | "*)").

25.* *lexical-directive* = "%include" "'" *file-name-string* "'" ";" |
"%page" ";" .

Pascal-2 Syntax

This set of productions defines the syntax for the language accepted by the Pascal-2 compiler, including all extensions. This section is to be interpreted in conjunction with the lexical description of the language.

Productions are based on those in the standard. Where the language accepted by the Pascal-2 compiler differs from this standard, the production is marked with an asterisk ("*").

1.* *program* = [*program-heading*] { *label-declaration-part*
| *constant-definition-part* | *type-definition-part*
| *variable-declaration-part* | *routine-declaration* } [
body "."] .

2. *program-heading* = "program" *identifier* ["(" *program-parameters* ")"] ";" .

3. *program-parameters* = *identifier* { "," *identifier* } .

4. *block* = *declarations body* .

5.* *declarations* = [*label-declaration-part*] [*constant-definition-part*]
[*type-definition-part*] [*variable-declaration-part*]
{ *routine-declaration* } .

6. *label-declaration-part* = "label" *label* { "," *label* } ";" .

7. *constant-definition-part* = "const" *constant-definition* { ";" *constant-definition* } ";" .

8. *constant-definition* = *identifier* "=" *constant* .

9.* *constant* = ([*sign*] (*unsigned-number* | *identifier*))
| *character-string* | *structured-constant* .

10.* structured-constant = structured-type-identifier "=" constant-component-list .

11.* constant-component-list = "(" constant-component { "," constant-component } ")" .

12.* constant-component = constant | constant-component-list .

13. type-definition-part = "type" type-definition { ";" type-definition } ";" .

14. type-definition = identifier "=" type .

15. type = identifier | enumerated-type | subrange-type | set-type | array-type | record-type | file-type | ("^" | "@" identifier) .

16. enumerated-type = "(" identifier { "," identifier } ")" .

17. subrange-type = constant ".." constant .

18. set-type = ["packed"] "set" "of" type .

19. array-type = ["packed"] "array" "[" type { "," type } "]" "of" type .

20. record-type = ["packed"] "record" field-list [";"] "end" .

21. field-list = (fixed-part [";" variant-part]) | variant-part .

22. fixed-part = record-section { ";" record-section } .

23. record-section = identifier { "," identifier } ":" type .

24. variant-part = "case" [identifier ":"] identifier "of" variant { ";" variant } .

25. variant = constant { "," constant } ":" "(" [field-list] [";"] ")" .

26. file-type = ["packed"] "file" "of" type .

27. variable-declaration-part = "var" variable-declaration ";" { variable-declaration ";" } .

28. variable-declaration = var-specification { "," var-specification } ":" type .

29. var-specification = identifier ["origin" constant] .

30. routine-declaration = (procedure-declaration | function-declaration) ";" .

31. *procedure-declaration* = (*procedure-heading* ";" *block*)
| (*procedure-heading* ";" *directive*) | (*procedure-ident* ";" *block*) .

32. *procedure-heading* = "**procedure**" *identifier* [*parameter-list*] .

33. *procedure-ident* = "**procedure**" *identifier* .

34. *function-declaration* = (*function-heading* ";" *block*)
| (*function-heading* ";" *directive*) | (*function-ident* ";" *block*) .

35. *function-heading* = "**function**" *identifier* [*parameter-list*] ":" *identifier* .

36. *function-ident* = "**function**" *identifier* .

37. *parameter-list* = "(" *parameter-section* { ";" *parameter-section* } ")" .

38. *parameter-section* = (["**var**"] *identifier* { "," *identifier* } ":" (*identifier*
| *conformant-array-schema*)) | *procedure-heading* | *function-heading* .

39. *conformant-array-schema* = *packed-conformant-array-schema*
| *unpacked-conformant-array-schema* .

40. *packed-conformant-array-schema* = "**packed**" "**array**" "[" *index-type-specification* "]" "**of**" *type* .

41. *unpacked-conformant-array-schema* = "**array**" "[" *index-type-specification*
{ ";" *index-type-specification* } "]" "**of**" (*type* | *conformant-array-schema*) .

42. *index-type-specification* = *bound-identifier* ".." *bound-identifier* ":" *type* .

43. *body* = *compound-statement* .

44. *statement* = [*label* ":"]
[*assignment* | *procedure-call* | *compound-statement* | *if-statement*
| *case-statement* | *while-statement* | *repeat-statement*
| *for-statement* | *with-statement* | *goto-statement*] .

45. *assignment* = *variable* ":=" *expression* .

46. *procedure-call* = *identifier* [*arg-list* | *write-arg-list*] .

47. *arg-list* = "(" *expression* { "," *expression* } ")" .

48. write-arg-list = "(" write-arg { "," write-arg } ")" .

49. write-arg = expression [":" expression [":" expression]] .

50. compound-statement = "begin" statement { ";" statement }
"end" .

51. if-statement = "if" expression "then" statement ["else" statement] .

52.* case-statement = "case" expression "of" [case-element { ";"
case-element }] [";"]
["otherwise" statement [";"]] "end" .

53. case-element = constant { "," constant } ":" statement .

54. while-statement = "while" expression "do" statement .

55. repeat-statement = "repeat" statement { ";" statement } "until" expression .

56. for-statement = "for" identifier ":=" expression ("to" |
"downto") expression
"do" statement .

57. with-statement = "with" expression { "," expression } "do"
statement .

58. goto-statement = "goto" label .

59. expression = simple-expression [relational-operator simple-expression] .

60. relational-operator = "<" | ">" | "<=" | ">=" | "=" | "<>" | "in" .

61. simple-expression = [sign] term { adding-operator term } .

62. adding-operator = "+" | "-" | "or" .

63. term = factor { multiplying-operator factor } .

64. multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

65. factor = unsigned-constant | variable | function-call | "not" factor
| "(" expression ")" | bound-identifier | ("[" | "(."
)
[member-designator { "," member-designator }] (
"]" | ".)") .

66. unsigned-constant = unsigned-number | string | identifier | "nil"
.

67. function-call = identifier [arg-list] .

68. *variable* = *identifier* | *variable* (“[” | “(.”) *expression* { “,” *ex-
pression* }
(“]” | “.)”) | *variable* (“^” | “@”) | *variable* “.”
identifier .

69. *member-designator* = *expression* [“..” *expression*] .

Pascal–2 V2.1/RSX Debugger Guide

Debugging tools help uncover "run-time" errors—errors in a program's execution—that cannot be caught during compilation. For example, a procedure may generate an incorrect number of loops or make a legal but unintended change in the value of a variable. The Pascal–2 Debugger lets you control a program's execution interactively; you may suspend execution at particular statements to examine or modify the values of variables, or you may execute statements one at a time to trace the actions leading to an incorrect result.

When called, the Pascal–2 Debugger keeps track of all constants, variables, local procedures and functions and all standard and user-defined data types. The Debugger can show what's happening to data and allow you to change the data as the program executes. You can display the original source text of your program for immediate identification of context, and you can access and debug external procedures and functions called by the main program. (See "Debugging External Modules" at the end of this guide for details.) The Debugger also traps errors by halting execution of a program at the point of breakdown and identifying the last statement executed. Taken together, these features allow you to trouble-shoot a program until you have detected and corrected any errors.

This guide serves as an introduction to the Pascal–2 debugging process and as a comprehensive resource for operation of the Pascal–2 Debugger. The guide provides:

- An overview of the Pascal–2 debugging environment.
- Detailed descriptions of the Debugger commands.
- A tutorial that demonstrates the context in which Pascal–2 Debugger commands are most frequently used.
- An explanation of how external modules are debugged.
- A one-page summary of Debugger commands.

A word of warning before beginning: specifying the debugging option causes the compiler to include a call to the Debugger before each procedure and statement, which substantially increases the size of your program. The object module created by the compiler contains extra code to locate statements and procedures in your program. Moreover, introducing the Debugger turns off optimizations that would interfere with debugging. The compiler normally folds similar statements into one section of code and optimizes the usage of some variables by keeping their values in registers on the stack temporarily. These optimizations would keep the Debugger from setting breakpoints in statements and from changing the values of variables while your program was running—both of which are important debugging facilities. A little bit of memory is saved during use of the Debugger by disabling the procedure walkback—this happens automatically when the Debugger is implemented—but in general, the overhead involved in using the Pascal-2 Debugger is about one word per Pascal statement and about six words per procedure. Code returns to its normal size once you correct any problems and recompile your program without a call to the Debugger. (See "Overlays" in this guide regarding what to do if the program grows too large.)

CAUTION

The Debugger cannot be used on I & D space programs. The separation of instructions and data generated by such programs makes it difficult for the Debugger to properly trace procedures and statements. (See "Run-Time Error Reporting" for more details.)

Including the Pascal-2 Debugger in Your Program

The **debug** compilation switch invokes the Pascal-2 Debugger. (See the User Guide for details on compilation switches.) Using the **debug** switch in your compilation command automatically generates a formatted listing file, with an .LST extension, in the same directory as the output file. The Debugger reads this listing file to display the source lines when statements are identified. The Debugger can use only the listing file produced by the **debug** switch.

The **debug** switch also causes the compiler to create symbol table and statement map files in the same directory as the output file. The symbol table file (extension .SYM) describes the constants, types, variables, and the memory layout of variables. The symbol table file also contains information about each procedure and function local to the compilation unit. The statement map file (extension .SMP) contains a map of the location of the statements and their position in the listing. Both files are in binary form and are not readily examined by users.

After correcting any syntax errors discovered in a normal compilation and then compiling the program with **debug**, you must task build the resulting object file (and any external procedures or functions that the program calls) with the Pascal support library. The support library contains all required Pascal-2 Debugger routines.

For the sample program ROTAT.PAS, the proper compilation and task-build commands are given as:

```
>PAS ROTAT/DEBUG
>TKB
TKB>ROTAT/CP/FP=ROTAT,LB:[1,1]PASLIB/LB
TKB>/
Enter Options
TKB>UNITS=20
TKB>//
```

Identifying Pascal Statements

Remember, the **debug** switch automatically generates a listing file. As the example ROTAT.LST shows, a listing file has two columns of numbers. The leftmost column lists the line numbers in the source file. The second column contains the number of each statement in the program, beginning with '1' for each procedure or function. These numbers identify points where you may set breakpoints to interrupt program execution. In ROTAT.LST, several lines accessible to the Debugger have been labeled by procedure name and statement number. As shown, statements in the main body of the program are considered to be in the procedure **MAIN**. All Pascal programs begin executing at **MAIN,1**.

You should have a printout of the listing file as reference when you begin a debugging session, or you may use the **L** command to list parts of the program while you are debugging.

Example: Listing for Program ROTATE.PAS

```
Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM    Site #1-1    Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 ROTAT/DEBUG
 Line Stmt
    1        program Rotat; { rotate an array of numbers }
    2
    3          const Arraylen = 7;
    4
    5          type  Index = 1..Arraylen;  Element = 0..10;
    6                Numbers = array [index] of Element;
    7
    8          var   I: Index;  N: Numbers;  Left, Right: Index;
    9
   10          procedure Rotate(First, Last: Index;
   11                            var A: Numbers);
   12            var  I: Index;
   13
   14    1       begin
   15    2         for I := First to Last do
  '16    3           A[I] := A[I + 1]; ------------------ Rotate,3
   17    4         A[Last] := A[First]; ----------------- Rotate,4
   18    5         write('Rotated ', first: 1, ' thru ', last: 1, '=');
   19            end;
   20
   21    1     begin { main program } ----------------- Main,1
   22    2       for I := 1 to Arraylen do
   23    3         begin  N[I] := I;  write(I: 2);  end;
   24    5       writeln;  write('Left,Right? ');
   25    7       readln(Left, Right);
   26    8       I := 4;
   27    9       Rotate(Left, Right, N);
   28   10       for I := 1 to Arraylen do
   29   11         write(N[I]: 2);
   30   12       writeln
   31          end.

*** No lines with errors detected ***
```

ROTAT.PAS is worth studying for a moment because it appears frequently throughout the remainder of this guide. The program prints an array of seven integers, then prompts you, asking for a starting and ending point in the array. Once the two input numbers have been entered, the program is supposed to rotate that section of integers to the left, with the left digit replacing the right. In its current form, the program compiles without problem, but as is shown in several of the following sections, its execution encounters numerous run-time errors.

Controlling the Debugger

After compiling and task building the program, you can now run it. The Debugger takes control of the program, enters the command mode, and prompts with a right brace '}' symbol. (This may print on upper-case-only terminals as the right bracket ']' character.)

```
>RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

}
```

You control the Debugger through single-character commands that generally take one of two forms, depending on whether or not the command accepts parameters:

} *single-character command*
} *single-character command* (*parameter(s)*)

Debugger commands and their parameters may be typed in either upper or lower case.

Command Syntax

In general, Debugger commands are used for controlling breakpoints, program execution, program tracking, data, and for displaying information about the data being maintained by the Debugger. Debugger commands can be stored in series and executed at designated locations within a program. Such locations are known as breakpoints and are specified by the break command. At any breakpoint, you may enter as many stored commands as fit on a single line. Any Debugger command may appear in a stored command, and certain utility commands, described later, allow macros to be defined that let you store combined commands.

As an example of the general use of Debugger commands and the syntax for writing stored commands, look at the following command line. The line begins with the single-character command `B` followed by the parameter `ROTATE,3`. These direct the Debugger to set a breakpoint at statement 3 in procedure `ROTATE`. Next, a stored command is used to instruct the Debugger to write (`W`) the values of the variables `I` and `A[I]`. Stored commands are specified by placing them within angle brackets (<...>) after a break command and separating them by semicolons.

```
} B(ROTATE,3) <W(I); W(A[I])>
```

The Debugger accepts any of the single-character commands defined in the following sections. Numeric parameters in these sections are indicated by '*n*', as in the command `S(n)`. A summary of the Debugger commands is given in Appendix A at the end of this guide, and the `?` (question mark) command prints a similar list on your terminal screen.

Exiting and Stopping the Debugger

To exit from the Debugger at the prompt, give the command **Q** (quit), or type a Control-Z (**^Z**), or type Control-C (**^C**) twice in a row. A single Control-C (**^C**) typed during program execution stops the Debugger, thus permitting you to break into "infinite loops" in your program.

Selective Debugging

For certain large programs, you may wish to selectively debug portions of a program in order to speed up the debugging process or to reduce the amount of memory overhead created by the Debugger. You can edit your program to turn off generation of debugging information around procedures that have already been tested and debugged by using the embedded directives **$nodebug** and **$debug**. To turn off debugging, place the directive **$nodebug** before the procedure definition and the directive **$debug** after the procedure. **$Nodebug** and **$debug** are effective only when the **/debug** switch is first specified in the compilation command line. Otherwise they are ignored by the compiler. (See the User Guide for further details on embedded directives.)

Breakpoint Commands

Breakpoint commands allow you to set or remove breakpoints when your program reaches a certain point in execution or when a specified variable in your program changes value. Breakpoints allow you to interrupt the program in order to execute other Debugger commands.

B, B(): Control Breakpoints

A program control breakpoint is identified by two items: a block name (procedure, function, or MAIN), and a statement number within that block. For example, ROTATE,3 identifies the third statement in the procedure ROTATE. Statements are sequentially numbered within each block. Statement numbers are listed in the second column of the program listing produced by the debug switch.

The B(*block*,*stmtnum*) command sets a control breakpoint within the block named *block* at the statement numbered *stmtnum*. When the breakpoint is reached, your program is interrupted before execution of the named statement, the breakpoint is identified, and the Pascal source line is displayed. The Debugger then accepts commands.

These may be interactive commands (from your terminal) or stored commands executed automatically. Any Debugger command may be stored for execution at a breakpoint. Stored commands are executed before interactive commands. If the stored commands direct the Debugger to resume execution, the program continues without waiting for an interactive command.

You may interrupt the program at any time with a Control-C (^C). This command stops the program and identifies the point of interruption as if you had set a control breakpoint.

NOTE

If you type a Control-C (^C) while the program is awaiting input for a real or an integer at a read or readln statement, the Control-C (^C) does not take effect until after you have completed the input request.

A run-time error or program termination also causes a control breakpoint after the error message or termination status is displayed. You may set any number of control breakpoints. (The program executes more slowly if you define many.)

To set breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

You may remove a breakpoint in two ways. The B command with no parameters deletes the breakpoint that most recently stopped the program. Otherwise, the K command described next may be used. (For uses of the B command, see the example listed after the C command.)

K, K(): Killing of Breakpoints

The K(*block*,*stmtnum*) command deletes the breakpoint specified by its parameter; the K command with no parameters removes all breakpoints. (See the example after the C command.) To remove breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

V, V(): Data Breakpoints (Variables)

The data breakpoint facility (also called the "watched variable" command) causes an immediate breakpoint when the value of a specified variable is changed. The V(*variable*) command sets a data breakpoint, with *variable* indicating the variable to be monitored. When the value of the variable is changed, the Debugger prints both the old and new values and interrupts program execution for commands. Like control breakpoints, data breakpoints may have stored commands that are automatically executed when the breakpoint is triggered. A list of the stored commands, separated by semicolons, is enclosed in angle brackets after the watched variable command: V(*variable*)< ... >.

The V command monitors a variable of any type, but only the first 32 bytes of data is watched. You may watch any number of variables. (The program executes slowly if you set many.) For variables defined locally to a procedure, the watch command can either be set from within the procedure or through use of the E command defined later in this guide.

```
} B(ROTATE,1)<V(A[6])> ——————————— variable watch set within ROTATE
} G———————————————————————————— begin execution
Left,Right? 2 6———————————————— input to ROTAT
Breakpoint at ROTATE,1   begin
} C———————————————————————————— continue execution
The value of "A[6]" was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4   A[Last] := A[First];
}
```

If a local variable is being monitored and the associated block is completed, the Debugger removes the breakpoint and displays a message that the variable no longer exists.

```
Breakpoint at ROTATE,1  begin ──────────── previously set breakpoint
} L ─────────────────────── lists statements of procedure ROTATE
   14    1      begin
   15    2        for I := First to Last do
   16    3          A[I] := A[I + 1];
   17    4        A[Last] := A[First];
   18    5        write('Rotated ', first: 1, ' thru ', last: 1, '=');
   19            end;
} V(A[2]) ────────────────── variable watched within ROTATE
} C
The value of ''A[2]'' was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 2
New value: 3
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} C
Watch terminated for ''A[2]'' Value did not change.
Rotated 2 thru 6= 1 3 4 5 6 3 7 ──────────── indicates a run-time error
}
```

This example gives us our first indication of a problem in the program ROTAT.PAS.The correct rotation for the starting and ending points (2,6) should read 1 3 4 5 6 2 7 not 1 3 4 5 6 3 7. Correction of the problem is explained in the section "Stepping Through a Debugging Session" later in this guide.

The V command without parameters removes all data breakpoints. It is not possible to remove individual data breakpoints.

Execution Control Commands

Execution control commands provide the means to monitor and control the flow of the program. The commands initiate, interrupt, or continue execution.

G: Go

The G (Go) command begins executing the program at MAIN,1 and may be used at any point in the program to restart it.

See the example after the C command.

C, C(): Continue Execution

The C (Continue) command resumes program execution from the current breakpoint.

If you set a breakpoint inside a loop, you may use the C(*n*) command to let the statement at the breakpoint execute *n* times. For instance, you may set a breakpoint at COUNT,10 inside a loop structure. When the Debugger stops at that breakpoint, you may give the command C(6) to let the loop iterate six times before the program stops again at COUNT,10. Each breakpoint has its own counter, which is independent of the counters for other breakpoints.

The C command functions like the G command to begin executing the program if you are at the start of the program.

If you use the C command after the program has terminated, you receive an error message telling you to use the G command to restart the program.

Example: Use of the B, K, D, G and C Commands

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(main,8,2)———————————— List 8th statement of MAIN, 2 lines
   26     8      I := 4;
   27     9      Rotate(Left, Right, N);
} B(main,9)<W('I=',i);C>
} B(Rotate,3)<W('In rotate, I=',i)>
} G
 1 2 3 4 5 6 7
Left,Right? 2 6———————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
I=  4
Breakpoint at ROTATE,3  A[I] := A[I + 1];
In rotate, I=  2
} D———————————————————————————— display breakpoints

Breakpoints

ROTATE,3  A[I] := A[I + 1];
      <W('In rotate, I=',I)>

MAIN,9  Rotate(Left, Right, N);
      <W('I=',I);C>
} W(I); C(2); W(I)
2
Breakpoint at ROTATE,3  A[I] := A[I + 1];
In rotate, I=  4
4
} K(Rotate,3)———————————————————— kill specified breakpoint
} C———————————————————————————— continue execution
Rotated 2 thru 6= 1 3 4 5 6 3 7———————— indicates run-time error
} D———————————————————————————— display breakpoints

Breakpoints

MAIN,9  Rotate(Left, Right, N);
      <W('I=',I);C>

} K———————————————————————————— kill all breakpoints
} D—————————————————————————— (no breakpoints to display)
} Q———————————————————————————— quit the Debugger
```

S, S(): Step to Next Statement

The S (Step) command executes the next statement of the program. The S(n) command executes n statements without interruption. If a statement being "stepped" calls another procedure or function, that new procedure or function also is executed one step at a time.

See the example after the P command.

P, P(): Proceed to Next Statement

The P (Proceed) command executes the next statement at the current level of the program. P differs from S in that P does not single-step through functions and procedures called by the current statement. P treats an entire nested call as a single statement; thus procedure calls and function invocations are completed before program control returns to the Debugger, allowing you to bypass the detailed execution of routines (e.g., ones already debugged). If the current procedure ends, P begins single-stepping the procedure that called the current procedure.

The P(n) command is equivalent to repeating the P command n times.

As with the C command, you may not go past the end of the program with an S or a P command. If you do so, you receive an error message telling you to use G to restart the program.

Example: Use of the S and P Commands

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983
Debugging program ROTAT

} B(main,9)
} G
 1 2 3 4 5 6 7
Left,Right? 1 5 ———————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} S
Breakpoint at ROTATE,1  begin;
} S
Breakpoint at ROTATE,2  for I := First to Last do
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} S(4)
Breakpoint at ROTATE,4  A[Last] := A[First];
} C
```

```
Rotated 1 thru 5= 2 3 4 5 2 6 7———————— further indication of run-time error
} G
 1 2 3 4 5 6 7
Left,Right? 1 5—————————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} P
Breakpoint at MAIN,10 for I := 1 to Arraylen do
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P(6)
Rotated 1 thru 5= 2 3 4 5 2 6 7—————————— same indication of a problem
}
```

Tracking Commands

Two commands help you track program execution. The H command lists the statements that have brought you to your present position. The T command traces program execution through each statement.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the H command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the H command shows the steps leading to the statement causing the error. The H command with no parameters prints a list of the last 10 statements executed. H(*n*) prints the last *n* statements up to 50.

The H command has other important functions as well. See "Execution Stack Commands" for details and for examples of the command.

T(): Execution Trace

The T command accepts a Boolean parameter, either enabling or disabling the tracing of program execution. When tracing is enabled with the T(TRUE) command, each statement is identified by its block name and statement number and is displayed before being executed.

A Control-C (^C) interrupts the trace and returns the Debugger to command mode. You may then turn off tracing with the T(FALSE) command and continue running your program with the C command.

Example: Use of the T Command

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(main,9,3)
   27   9     Rotate(Left, Right, N);
   28   10    for I := 1 to Arraylen do
   29   11      write(N[I]:2);
} B(main,9)—————————————————————— set breakpoint
} G
 1 2 3 4 5 6 7
Left,Right? 1 3———————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} T(TRUE)
} C
ROTATE,1  begin  ——————————————————— tracing output begins
ROTATE,2  for I := First to Last do
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,4  A[Last] := A[First];
ROTATE,5  write('Rotated ',first: 1,' thru ',last: 1,'=');
MAIN,10  for I := 1 to Arraylen do
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,11  write(N[I]:2);
MAIN,12  writeln
Rotated 1 thru 3= 2 3 2 4 5 6 7———————— our run-time error is still evident
} T(FALSE) ————————————————————————— tracing off
} R
} G
 1 2 3 4 5 6 7
Left,Right? 1 3———————————————————— input to ROTAT
Rotated 1 thru 3= 2 3 2 4 5 6 7
}
```

Data Commands

Debugger data commands allow you to display the current values of variables and to assign new values to them. The data commands provide full access to user identifiers and type definitions. The data commands conform to Pascal type compatibility rules.

W(): Write Variable Value

You use the W command to write the value of a variable (including a pointer), of a constant, or of a memory location. The format for the W command is:

```
} W(name1,name2,name3,  ...)
```

where *name* is the name of the variable you want written. As shown, you may write the value of more than one variable by separating variable names with commas.

The type of variable determines the format of the output. For example, integers are displayed as signed decimal numbers. Set variables are displayed in Pascal set notation. Scalar variables are displayed as the names of the enumerated types they represent.

You may use the Pascal colon notation ':' to alter the way variables are written. For example, to print the integer variable I as a hexadecimal number, you use:

```
} W(I:-1)
```

Also see the example after Variable Assignment.

Real numbers may be formatted according to the same rules used by the compiler.

A numeric constant is used as an address if you wish to write the integer value contained in a memory location. A 'B' placed after the number, as in W(27740B), specifies an octal memory location. Memory locations are displayed as signed integers.

The Debugger may write any complex Pascal data structure, including records and arrays, except files.

The Debugger displays an array in an orderly fashion that reflects the array's structure. For each change in the least significant (rightmost) index of the array, the Debugger writes a space between elements. For each change in the next least significant (second-from-rightmost) index, the Debugger starts a new line. And for changes in the *nth* index, where *n* is the number of "places from the right" of the least significant index and *n* is greater than 2, the Debugger writes $n - 2$ blank lines and indents the first row of the display $n - 2$ spaces.

Example: Debugger's Display of Array's Structure

```
Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM    Site #1-1    Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 ROTAT/DEBUG

 Line Stmt
    1       program Multi;                       { multidimensional variables }
    2
    3       var A: array [1..3, 1..3, 1..3] of integer;
    4           I, J, K: integer;
    5
    6     1 begin
    7     2   for I := 1 to 3 do
    8     3     for J := 1 to 3 do
    9     4       for K := 1 to 3 do
   10     5         A[I,J,K] := (I * 10 + J) * 10 + K;
   11       end.

 *** No lines with errors detected ***



>RUN MULTI


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program MULTI

} G

Program terminated.

Breakpoint at MAIN,5  A[I,J,K] := (I * 10 + J) * 10 + K;
} W(A)
 111 112 113
121 122 123
131 132 133

 211 212 213
221 222 223
231 232 233

 311 312 313
321 322 323
331 332 333
```

When you write records, the Debugger lists each field name followed by the value of that field. The format of each field is determined by the data type of the field. Complex records, such as those containing arrays of records, can get messy; you may want to have the listing on hand to show the definition of the record being printed.

Variable Assignment

The Debugger command to modify the value of a program variable is identical in form to a Pascal assignment statement. The left-hand side of the ':=' assignment operator indicates the variable to be modified. This variable may include array indices, record field selectors, and pointer accesses. The right-hand side specifies the value to be assigned. This may be a simple constant or literal value, or another program variable. Standard notation is used for all values, including sets. General expressions (operators and functions) are not permitted.

Debugger variable assignments must conform to the Pascal assignment compatibility rules. All variables accessed in an assignment command must be available in the current stack context. The E(*n*) command may be used to temporarily change context, if necessary.

Example: Use of the W Command and Variable Assignment

```
Pascal-2 RSX V2.1E   9-Feb-86   7:06 AM     Site #1-1     Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 COLOR/DEBUG
 Line Stmt
    1       program Color;
    2
    3         type
    4           Color = (Red, Orange, Yellow, Blue, Green);
    5
    6         var
    7           c: Color; I: integer;
    8           Colorset: set of Color;
    9           a: array [0..4] of Color;
   10           r: record
   11               I: integer;
   12               S: set of Color;
   13               C: packed array [1..4] of char;
   14             end;
   15
   16     1   begin
   17     2     for C := Red to Green do A[ord(C)] := C;
   18     4     Colorset := [Red, Yellow..Green];
   19     5     R.I := 123; R.S := [Orange, Green]; R.C := 'TEST';
   20         end.

*** No lines with errors detected ***
```

```
>RUN COLOR
```

```
Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program COLOR

} G
} W(A)
 RED ORANGE YELLOW BLUE GREEN

} A[1] := Red; A[4] := Red; W(A)
 RED RED YELLOW BLUE RED

} W(Colorset)
[RED,YELLOW..GREEN]
} Colorset := [Red..Green]; W(Colorset)
[RED..GREEN]
} W(R)
I: 123
S: [ORANGE,GREEN]
C: TEST

} R.I := 321; R.S := Colorset; W(R)
I: 321
S: [RED..GREEN]
C: TEST

}
```

Informational Commands

Informational commands show data being maintained by the Debugger. The D command shows the current breakpoints, user-defined macros, and variables being watched. The L command shows selected parts of the program listing, so that you won't have to reprint the listing each time you revise your program.

D: Display Parameters

The D command displays all breakpoints, user-defined macros, and the variables being watched; it also shows any commands associated with each. Breakpoints are set with the B command. Macros are stored Debugger commands created by the M command and executed by the X command. The V command is used to set variable watches. (See the respective sections for details on these commands.)

See the ROTAT example in "Running the Debugger" and the example after the C command.

L, L(): List Source Lines

The L command uses the statement numbers in the listing file of your program to list portions of the source program. The L command allows you to list individual statements, parts of procedures, or entire procedures.

When a breakpoint is set at a statement with B(), the Debugger prints only the first line associated with the statement. The History command H also prints only the first line of the statement. The L command, in contrast, prints all lines containing the statement.

The L command with no parameters lists the current procedure. You may list any other procedure by giving the procedure name enclosed in parentheses. For example, L(MAIN) lists the body of the main program.

The command L(*proc*,*stmtnum*) lists a single statement, where *proc* is the name of the procedure and *stmtnum* is the number of the statement to print.

You also may list sections of the program starting or ending at a particular statement by specifying a line count after the statement number. For example, L(MAIN,1,10) lists the first ten lines of the main program.

The general form of the command is:

```
} L(proc,stmtnum,count)
```

where *proc* and *stmtnum* describe a statement in the program. A positive *count* prints that many lines starting at the statement specified and moving forward. A negative *count* causes the Debugger to list statements up to and including *stmtnum*. (The listing of source lines in external functions and procedures requires a slightly different form of the L command. See "Debugging External Modules" later in this guide for details.)

This example lists 5 lines beginning with the first statement of procedure ROTATE:

```
} L(Rotate,1,5)
   14    1       begin
   15    2         for I := First to Last do
   16    3           A[I] := A[I + 1];
   17    4         A[Last] := A[First];
   18    5         write('Rotated ',first: 1,' thru ',last: 1,'=');
```

This example lists 2 lines leading up to and including the 4th statement of procedure Rotate:

```
} L(Rotate,4,-2)
   16    3           A[I] := A[I + 1];
   17    4         A[Last] := A[First];
```

When you list an entire procedure, the Debugger attempts to include the procedure heading and local variable declarations in the listing. However, this header information is only used by the Pascal compiler, so the Debugger has to estimate where the procedure header information is located in the listing file. As a result, the Debugger may not always print the complete header information or may sometimes print part of the preceding procedure.

Long procedures may take some time to print. A single Control-C (^C) interrupts the listing and returns the Debugger to command mode.

Utility Commands

Utility commands allow you to define a series of commands as a macro to be executed by entering a single command.

M(): Define Macro

The M command saves you some typing when you need to issue repetitive commands. For example, you may need to write the value of several critical variables at different places in your program. The M feature lets you combine these commands under one name, then execute this group of commands by using the X command, explained below. You cannot pass parameters to macros.

The format for definition of a macro is:

} M(*name*)<*command1*; *command2*; *command3*; ...>

where *name* is any alphanumeric string containing up to 32 symbols. The X command uses *name* to identify the macro. You may place as many Debugger commands in the angle brackets '< >' as fit on one command line. You may delete a macro by typing M(*name*) with no commands. Available memory is the only limit on the number of macros you may define. The D command lists macro names and the commands associated with each name.

See the example after the X command.

X(): Execute Macro

You may execute the Debugger commands associated with a macro by using the X command. The format is:

```
} X(name)
```

where *name* is the name of the macro. The effect of the X command is to execute the Debugger commands defined by the M command of that name.

Example: Use of the M and X Commands

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} M(DumpN)<W('The value of N=',N)> ——————————— define macro
} B(Rotate,1); G
 1 2 3 4 5 6 7
Left,Right? 2 6 ——————————————————— input to ROTAT
Breakpoint at ROTATE,1  begin
} M(DumpI)<W('I=',I)> ————————————————— define macro
} D
Breakpoints

ROTATE,1  begin

Macros

DUMPI      W('I=',I)
DUMPN      W('The value of N=',N)
} S
Breakpoint at ROTATE,2  for I := First to Last do
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} X(DumpI) ———————————————————————————— execute macro
I=  2
} S(4); X(DumpI); X(DumpN)
Breakpoint at ROTATE,3  A[I] := A[I + 1];
I=  6
The value of N=   1 3 4 5 6 3 7

} Q
```

Execution Stack Commands

The execution stack commands allow you to trace down run-time errors by examining the stack. The H command shows at any time a history of program execution and the current stack of active procedure and function calls. The [illegible] command lists the names of the parameters and local variables in any procedure in the execution stack. The E command allows you to change the context of the stack frame from the current procedure to another so you may access variables you otherwise wouldn't be able to.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the H command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the H command shows the steps leading to the statement causing the error. The H command with no parameters prints a list of the last 10 statements executed. H(*n*) prints the last *n* statements up to 50.

The H command also lists the execution stack. Each time a procedure or function is called, a new entry is made at the top of the execution stack. When the procedure exits, that entry is removed from the top of the stack. The main program is always at the bottom of the stack. The H command shows the procedures that were called to get from the main program to the current procedure. H(0) prints just the execution stack.

In the display, each procedure or function in the execution stack is identified by a number. This procedure number is used to identify procedures in the execution stack for the [illegible] and E commands described in following sections. (These are not the statement numbers used to identify other Debugger commands.)

In the display, the '<' character marks the current procedure. Unless the E command is used the current procedure is always the top procedure in the execution stack. The Debugger uses the current procedure to determine the local variables that can be accessed according to Pascal scope rules. Procedures marked with the asterisk '*' character are those procedures that contain the lexical definition of the current procedure. The parameters and local variables in the procedures marked by '<' or '*' are the only local variables that you may look at or change directly. If you wish to look at local variables in other procedures in the execution stack, you must use the E command.

See the example after the E command.

N, N(): Names of Variables

The N command with no parameters lists the names of the parameters and local variables in the current procedure. If you are in the main program, the command displays all of the global-level variable names.

N with a numeric parameter lists the names of the local variables in the procedure so numbered on the execution stack. These numbers are obtained via the H command, described above.

Note that N lists the names of the local variables and parameters in any procedure or function on the stack, not merely the ones marked with the '*'. However, you cannot write or change the values of variables unless they are in procedures or functions marked with the '*'.

The E command allows access to variables that you otherwise cannot access from the current procedure.

See the example after the E command.

E(): Enter Stack-Frame Context

The Debugger normally enforces Pascal scope rules. If you stop your program in the middle of a procedure, you may write or modify only those variables and parameters of the procedures that enclose the current procedure, as described in the section on the H command.

To look at or change local variables in procedures that are not accessible to the current procedure, the E command gets around the Pascal scope rules by temporarily changing the context of the current procedure.

The H command numbers the procedures in the execution stack. The main program is always 1, and procedures called from the main program are listed as 2, and so on. If you want to examine variables in procedure 5 in the current execution stack, and it is not marked with an '*' (and therefore not available to you from where you are), you use E(5) to temporarily enter the context of that procedure.

The E command affects only debugging commands that follow it on the same command line. For example, to print the value of the variable I in the procedure listed as 5, you type:

```
} E(5); W(I)
```

This command line makes procedure 5 the current procedure. Then, using the context of procedure 5, the Debugger prints the value of the variable I. At the end of the command line, the current procedure is changed back to the top procedure in the execution stack.

Because the N command allows you to list the names of variables in all the procedures on the execution stack, the following commands are equivalent:

```
} E(5); N
} N(5)
```

Example: Use of the H, N, and E Commands

```
Breakpoint at CHECK,1  begin   { start of check }
} H(5) ———————————————— list last 5 statements executed

Program execution history:

ANALYZEMOVE,9  Vacant[Target] := false;
ANALYZEMOVE,10  if CentralSquares[Target] then
ANALYZEMOVE,14  PossibleMoves := PossibleMoves+1;
ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
CHECK,1  begin   { start of check }

Procedure execution stack

 8< CHECK,1  begin   { start of check }
 7* ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
 6* ANALYZE,12  AnalyzeMove(4,I); AnalyzeMove(5,I);
 5* EVALUATEBOARD,4  Analyze;
 4  GENMOVE,15  EvaluateBoard(N^,Turn);
 3  MOVEPIECE,9  if MovesAllowed then GenMove(I,J);
 2  EXPAND,11  if Color[Who]=Turn then MovePiece(I,I,0,0);
 1* MAIN,8  Expand(Root,True);

} N——————————————————————— local names
DIRECTION SRC DST F
} N(7)—————————————————— names in frame 7
DIRECTION I SAFE WASKING TARGET THRT
} N(4)—————————————————— names in frame 4
I J N OLDPIECE
} E(7); W(I)————————— change context to frame 7, write value
14
} E(4); W(I)————————— change context to frame 4, write value
27
} E(4); H(0)

Procedure execution stack

 8  CHECK,1  begin   { start of check }
 7  ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
 6  ANALYZE,12  AnalyzeMove(4,I); AnalyzeMove(5,I);
 5  EVALUATEBOARD,4  Analyze;
 4< GENMOVE,15  EvaluateBoard(N^,Turn);
 3* MOVEPIECE,9  if MovesAllowed then GenMove(I,J);
 2* EXPAND,11  if Color[Who]=Turn then MovePiece(I,I,0,0);
 1* MAIN,8  Expand(Root,True);

}
```

Stepping Through a Debugger Session

You seldom use only a single Debugger command at any one session, so no single-command example can demonstrate the context in which certain commands are used nor can it demonstrate all of the ways in which certain commands relate. Our approach, therefore, is to step through a sample program to demonstrate some of the common commands in a problem/example context.

In previous sections of this guide, several examples of run-time errors occurring in the execution of ROTAT were demonstrated. Let's begin this debugging session by compiling, task building and running the program and taking a closer look at what is going wrong.

```
>PAS ROTAT/DEBUG
>TKB
TKB>ROTAT/CP/FP=ROTAT,LB:[1,1]PASLIB/LB
TKB>/
Enter Options
TKB>UNITS=20
TKB>//
>RUN ROTAT
```

After compiling and building the program, you can now run it. The Debugger takes control of the program and enters command mode.

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

}
```

When the Debugger has taken control of the program, we instruct it to go (G), then we enter the starting point (2,6).

```
} G
 1 2 3 4 5 6 7
Left,Right? 2 6
Rotated 2 thru 6= 1 3 4 5 6 3 7
```

The problem we noted earlier persists. Our starting number in the rotation is apparently incremented by 1 each time the program is run. In this case, the second 3 in our rotated sequence should be 2. With the

L command, we now list the part of the main program that initializes the N array. From this, we can choose a location for a breakpoint once the array is initialized.

```
} L(main,1,5)—————————————————————— list 5 lines of main program
   21    1     begin    { Main program }
   22    2       for I := 1 to ArrayLen do
   23    3         begin N[I] := I; write(I:2); end;
   24    5       writeln; write('Left,Right? ');
   25    7       readln(Left, Right);
} B(main,6)———————————————————————— set breakpoint at MAIN,6
} G———————————————————————————————— begin execution
 1 2 3 4 5 6 7
Breakpoint at MAIN,6  writeln; write('Left,Right? ');
} W(N[6])—————————————————————————— write value of N[6]
6
```

(Note the way in which the Debugger counts statements when more than one is placed on a line, as on line number 24 above. Though not explicitly listed, the second statement on line 24 is statement number 6 and must be identified as such.)

Examination of the array N at this breakpoint shows the array to be correct; the change to the value of the variable must be occurring somewhere else. Using the V (watched variable) command, we tell the Debugger to stop the program whenever N[6] is changed.

```
} V(N[6])—————————————————————— watch for changes of value of N[6]
} C———————————————————————————— continue execution
Left,Right? 2 6———————————————— input to ROTAT
The value of "N[6]" was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4 A[Last] := A[First];
}
```

This is an expected change based on the algorithm being used in the rotation. Our last number is first incremented by 1 before being replaced by the first. Therefore, we continue to watch the variable.

```
} C
The value of "N[6]" was changed by the statement:
ROTATE,4  A[Last] := A[First];
Old value: 7
New value: 3
Breakpoint at ROTATE,5 write('Rotated ',first:1,' thru ',last:1,'=');
} W(First,Last) ——————————————— write values of First and Last
2 6
} W(N)————————————————————————— write values of array N
 1 3 4 5 6 3 7

} Q———————————————————————————— quit the Debugger
```

At ROTATE,4 the first element is assigned to the last element after the first element has already been changed. We must introduce a temporary variable to hold the first element value so that it is not destroyed. We correct the program (adding a "temp" variable, an assignment at line 14 and another between lines 16 and 17), then recompile with the /debug switch. Again, we use the L command to inspect the part of the program we changed.

```
>RUN ROTAT
```

```
Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(ROTATE,1,6) ——————————— list 6 lines of procedure ROTATE
   14   1       begin Temp := A[First];
   15   3         for I := First to Last do
   16   4           A[I] := A[I + 1];
   17   5         A[Last] := Temp;
   18   6         write('Rotated ',first: 1,' thru ',last: 1,'=');
   19            end;
} G————————————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 2 6———————————————— input to ROTAT
Rotated 2 thru 6= 1 3 4 5 6 2 7
} G————————————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 3 4———————————————— input to ROTAT
Rotated 3 thru 4= 1 2 4 3 5 6 7
Breakpoint at MAIN,12  writeln
```

Now the program seems to be running correctly, but let's make one more test before we've satisfied ourselves that the program is running as we want. Note that the G command restarts the program even after it has terminated.

```
} G————————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 1 7———————————— input to ROTAT

TT2 -- Fatal error at user PC=1424
TT2 -- Fatal error at user PC= 1424 Array subscript out of bound

}
```

The end points of a data subrange are always good places to look for run-time errors such as "Array subscript out of bounds," because they are the values most likely to exceed the predefined limits. We begin analyzing this new error by writing the values of variables found in the line where the error occurred.

```
} W(I)————————————— write the value of I
7
} W(A[8])——————————— write the value of A[8]
Array subscript too large
W(A[8])
    ^
The limits are 1..7
} Q———————————————— quit the Debugger
```

Now we can diagnose the error. We see that the limits for the array subscript of A have been exceeded by one. The `for` loop in the `ROTATE` procedure is likely to be looping too many times. We reduce the final value by 1 (`last` becomes `last-1` in line 15) and recompile the program. When we run the program, we tell the Debugger to list procedure `ROTATE`, so that we can more closely follow the section of the program we changed.

```
>RUN ROTAT
```

```
Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(ROTATE,1,6) ————————————— list 6 lines of procedure ROTATE
   14    1       begin Temp :=A[First];
   15    3         for I := First to Last - 1 do
   16    4           A[I] := A[I + 1];
   17    5         A[Last] := Temp;
   18    6         write('Rotated ',first: 1,' thru ',last: 1,'=');
} G——————————————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 1 7———————————————— input to ROTAT
Rotated 1 thru 7= 2 3 4 5 6 7 1
} Q ——————————————————————————— quit the Debugger
```

The results our program gives are now exactly what they should be. Once satisfied that the program is correct, we recompile it without the `debug` switch to reduce file and memory requirements, to improve execution speed, and to reinstate the walkback.

Debugging External Modules

An external module consists of one or more Pascal procedures or functions written and compiled independently of the main program that invokes it. The Debugger's ability to debug external modules allows you to fully debug an entire program, including externals, and also allows you to debug external procedures and functions only, in the context of a main program.

The debugging of external procedures and functions is simply a matter of compiling the module with the **debug** and **nomain** switches, linking the module with the main program, and upon execution, supplying the module name on certain Debugger commands (see below). The **debug** compilation creates the necessary symbol table files for the module. (Remember to compile the main program with the **debug** switch, too.) Refer to "External Modules" in the Programmer Reference for rules on the use of external modules.

As mentioned earlier, external modules cannot be debugged directly; they must be called from a main program. To debug an external procedure or function itself, create a short main program that simply invokes the procedure, then exits. Be sure to initialize any variables required of the call. Then compile the main program using the **debug** switch and task build it with the external module.

Differences in the Commands

This section covers only the differences in command syntax and usage; unless otherwise noted, the Debugger commands work as described in previous sections.

In general, the major differences are:

- The B, K and L commands accept the module name followed by a colon (:) as the first argument. These commands allow you to set and kill breakpoints and list source lines in external procedures and functions. The revised syntax for these commands are as follows:

```
} B(module:block,stmtnum)< ... >
} K(module:block,stmtnum)
} L(module:block,stmtnum,count)
```

- module name *module* is the name of the source file minus extension. For example, TEST is the module name for TEST.PAS. *Block* is the name of the procedure or function being referenced in *module*. The other arguments are the same as described in earlier sections.

- When displaying breakpoint and source-line information, the Debugger includes the module name along with the procedure name and line number. The D command, in addition to displaying breakpoints, user-defined macros, and variables being watched, shows you which module is currently being debugged.

- Defaults apply to the current module being debugged. To list lines, set breakpoints or kill breakpoints in any module but the current one, you must specify at least the module name and the procedure name on the commands.

- If you try to list lines or set/kill breakpoints in an external module not compiled for debugging, the run-time error "can't open file" causes the Debugger to abort trying to open the listing and symbol table files for that module. Of course, if you have old listing and symbol table files for that module lying around, the Debugger opens these files even if you did not wish to debug that module. In this case, if the data files do not match the source you're using, your results may not be accurate.

To illustrate the debugging of external modules, we present a single debugging session in which the above commands are used. In this example, the main program ROTAT.PAS, presented earlier, calls the external procedure **Rotate** contained in XROT.PAS. (In the previous program, procedure **Rotate** was a local procedure.)

After compiling the main program and external modules and task building them, run the program.

```
>RUN ROTAT


Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L——————————— defaults to main program
   14    1    begin { main program }
   15    2      for I := 1 to Arraylen do
   16    3        begin  N[I]  := I;  write(I: 2);  end;
   17    5      writeln;  write('Left,Right? ');
   18    7      readln(Left, Right);
   19    8      I := 4;
   20    9      Rotate(Left, Right, N);
   21   10      for I := 1 to Arraylen do
   22   11        write(N[I]: 2);
   23   12      writeln
   24         end.
} B(MAIN,7)
} B(XROT:ROTATE,5)<W(TEMP)>
```

Initially, the L command without parameters lists the main program by default because it is in the current module being debugged. The first breakpoint command could just as easily be B(ROTAT:MAIN,7). However, the module name is not necessary because the main program is the current module. The second breakpoint command shows that you must supply the module name for modules other than the current one. With the G command, program execution begins:

```
} G
 1 2 3 4 5 6 7
Breakpoint at ROTAT:MAIN,7  readln(Left, Right);
} C
Left,Right? 1 7————————— input to ROTAT
Breakpoint at XROT:ROTATE,5  A[Last] := Temp;
1
```

Note the way the Debugger reports the breakpoints. At this point the current procedure being debugged is the procedure Rotate, in external module XROT. (The single '1' is the value of Temp when the breakpoint is reached.) Now the B, X and L commands default to the external procedure Rotate, as shown below for the L command. To list the lines in the main program, you must specify the module name, as shown in the second L command:

```
} L
   16    1 begin  Temp := A[First];
   17    3   for I := First to Last-1 do
   18    4     A[I] := A[I + 1];
   19    5   A[Last] := Temp;
   20    6   write('Rotated ', first: 1, ' thru ', last: 1, '=');
   21      end;
} L(ROTAT:MAIN)
   14    1   begin { main program }
   15    2     for I := 1 to Arraylen do
   16    3       begin  N[I] := I;  write(I: 2);  end;
   17    5     writeln;  write('Left,Right? ');
   18    7     readln(Left, Right);
   19    8     I := 4;
   20    9     Rotate(Left, Right, N);
   21   10     for I := 1 to Arraylen do
   22   11       write(N[I]: 2);
   23   12     writeln
   24         end.
} D————————————— display current module and breakpoints

Current module: XROT

Breakpoints

XROT:ROTATE,5  A[Last] := Temp;
      <W(TEMP)>

ROTAT:MAIN,7  readln(Left, Right);

} H————————————————————— History command
```

```
Program execution history:

XROT:ROTATE,1   begin  Temp := A[First];
XROT:ROTATE,2   begin  Temp := A[First];
XROT:ROTATE,3   for I := First to Last-1 do
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,4   A[I] := A[I + 1];
XROT:ROTATE,5   A[Last] := Temp;

Procedure execution stack

 3< XROT:ROTATE,5   A[Last] := Temp;
 2* Module: XROT
 1* ROTAT:MAIN,9   Rotate(Left, Right, N);
```

Execution continues with the C command. The K and B commands used below demonstrate the rules governing the setting and removing of breakpoints. Note the erroneous breakpoint command and the corrected command.

```
} C———————————————— continue execution
Rotated 1 thru 7= 2 3 4 5 6 7 1

Program terminated.

Breakpoint at ROTAT:MAIN,12   writeln
} K(MAIN,7)————————————— kill breakpoint
} G
 1 2 3 4 5 6 7
Left,Right? 1 7——————————— input to ROTAT
Breakpoint at XROT:ROTATE,5   A[Last] := Temp;
1
} B(MAIN,7)————— won't work without the module name
No such statement in this procedure
B(MAIN,7)
    ^

} B(ROTAT:MAIN,7)—————————— that's better
} C
Rotated 1 thru 7= 2 3 4 5 6 7 1
} Q———————————————— quit the Debugger
```

Overlays

The Debugger adds up to 20K words to the size of a program, often making the task too big to run. If this happens, you must overlay the Debugger so that it takes less room. The file PAS.ODL overlays the Debugger modules in the Pascal support library (PASLIB) and the system I/O library (SYSLIB).

To build a task containing a program and the Debugger, you must create a short .ODL file that references PAS.ODL. For example, if the program is called TEST, you should create a file called TEST.ODL with the following contents:

```
@LB:[1,1]PAS.ODL
          .ROOT    ROOT-TEST,SYSIO,LIBR,DEBUG2
          .END
```

A more complicated example of an .ODL file is shown below. In this case, a main program, MAIN, calls the modules A, B, and C. To conserve space, A, B, and C overlay each other.

```
@LB:[1,1]PAS.ODL
USER:     .FCTR    TEST-(A,B,C)
          .ROOT    ROOT-USER,SYSIO,LIBR,DEBUG2
          .END
```

The factor called USER describes the way the modules in the program are overlaid. In this example, there is a main program called TEST.PAS, which is made up of modules A, B and C.

If the modules were placed in a library called MYLIB, you would create this .ODL file:

```
@LB:[1,1]PAS.ODL
MAIN:     .FCTR    MYLIB/LB:MAIN
A:        .FCTR    MYLIB/LB:A
B:        .FCTR    MYLIB/LB:B
C:        .FCTR    MYLIB/LB:C
USER:     .FCTR    TEST-(A,B,C)
          .ROOT    ROOT-MAIN-USER,SYSIO,LIBR,DEBUG2
          .END
```

Overlay descriptions may be arbitrarily complex, as long as the lowest-level factor is called USER.

The use of overlays greatly reduces the overhead required for the Debugger. On RSX systems with FPP floating-point hardware, the overhead for the Debugger, support library, and system I/O library is about 11K words, so you should be able to debug programs with up to 21K words of code and data.

A Task Builder command file can simplify the use of overlays with the Debugger. For example, if the .ODL file is called TEST.ODL, create a file called TEST.CMD with the following contents:

```
TEST/CP/FP,TEST=TEST/MP
UNITS=20
//
```

The command, `TKB @TEST`, will then task-build the program.

Appendix A: Debugger Command Summary

B	Remove current breakpoint
B(*block*,*stmtnum*)	Set a control breakpoint
B(*block*,*stmtnum*)< ... >	Control breakpoint with stored commands
PDP B(*module*:*block*, *stmtnum*)	Set a control breakpoint in external module
B(*module*:*block*,*stmtnum*)< ... >	Set external control breakpoint with stored commands
C	Continue program execution
C(*n*)	Continue *n* times
D	Display breakpoints and macros
E(*n*)	Enter context of frame *n*(1 line only)
G	Restart program
H(true/false)	Enable execution history
H	Display recent history and full stack
H(*n*)	Display last *n* statements executed
K	Remove all control breakpoints
K(*block*,*stmtnum*)	Remove specified breakpoint
K(*module*:*block*,*stmtnum*)	Remove specified breakpoint from external module
L(*proc*)	List source of *proc*
L(*proc*,*stmtnum*)	List statement *stmtnum* in *proc*
L(*proc*,*stmtnum*,x)	List x lines beginning with statement *stmtnum* in *proc*
L(*module*:*proc*)	List source of external module *proc*
L(*module*:*proc*,*stmtnum*)	List statement *stmtnum* in external module *proc*
L(*module*:*proc*,*stmtnum*,x)	List x lines beginning with statement *stmtnum* in external *proc*
M(*name*)<*commands*>	Define stored command macro
N	List variable names for current frame
N(*n*)	List variable names for frame *n*
P	Proceed 1 statement at current level
P(*n*)	Proceed *n* statements
Q	Quit Debugger
S	Single-step statement
S(*n*)	Single-step *n* statements
T(true/false)	Enable/disable tracing
V(*variable*)	Set data breakpoint
V(*variable*)< ... >	Data breakpoint with stored commands
W()	Write list of values
X(name)	Execute named macro command
variable := *value*	Assign *value* to *variable*
?	Help (display command summary)
^C (Control-C)	Immediate breakpoint
^Z (Control-Z)	Exit from Debugger

The Pascal-2 Profiler

The Pascal-2 Profiler can help you tune Pascal programs by detecting bottlenecks: small sections of code in which your program spends a disproportionately large amount of time. The Profiler counts the number of times each Pascal statement in your program is executed then prints a summary describing how many times each procedure is called and what percentage of the total statements executed are found in that procedure.

To use the Profiler, you should compile your program with the `profile` switch. (See the Programmer Reference for detailson compilation switches.) The `profile` switch causes the Pascal-2 compiler to generate several auxiliary files. These files, which permit the Profiler to locate the statements and procedures in your program, are the same ones generated by the `debug` switch. The Profiler requires 2K words of memory in addition to the space required by the program. In addition, program size increases about 4 words for each statement in the program. Thus, a program containing 1000 statements requires about 6K words for the Profiler.

After compiling your program you must task-build it with the Profiler, which is in the Pascal support library. To do this you must use the multiline form of the `TKB` command to increase the number of logical units available to the program (same as for the Debugger). The support library and Profiler open seven files, so you need at least 7 logical units to run a program that uses no files. (The system default is usually 6.) Although a smaller number may suffice, we recommend that you specify 20, to ensure that enough LUNs are available for execution.

The compilation and task-building steps are shown below, using a program, CHECKR.PAS, which plays a game of checkers.

```
>PAS CHECKR/PROFILE
>TKB
TKB>CHECKR/CP/FP=CHECKR,LB:[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=20
TKB>//
```

Upon execution, the Profiler takes control of the program and opens the program's auxiliary files created by the Pascal compiler. For large programs, there may be a short pause while the Profiler scans the auxiliary files to build internal data structures.

Next, the Profiler prompts for the name of the profile output file. If you specify a disk file, the default file extension is .PRO. Writing a profile to the terminal is practical only for a short program.

Compile and task build the program as previously shown, then run it:

```
>RUN CHECKR


profile  V2.1B -- 6-Feb-1983

Profiling module: CHECKR


Profile output file name? CHECKR— Output goes to CHECKR.PRO


Welcome to CHECKERS————— Program continues, slowly
```

The Profiler counts the number of times each statement is encountered. This counting of each statement slows down program execution. For this reason, it may not always be possible to profile programs that operate in a time-critical environment.

The Profiler generates a performance outline when the program terminates. Termination occurs when your program reaches the logical end of the program or when the program detects a fatal error condition. A Control-C (^C) interrupts the program and generates a profile at that point. Entering Control-C (^C) twice aborts the generation of the profile.

The Profiler listing has the same two columns of numbers as the Debugger listing (one column numbers each line of the source program and the other gives the statement number of the first statement on each line), plus an extra column of numbers at the far left of the listing.

This leftmost column lists the number of times the statement on that line is executed. If more than one statement appears on the line, the count applies only to the first statement on the line. To obtain an accurate count of each statement in the program, you can run your source program through the PASMAT formatter supplied with Pascal-2. The PASMAT 'S' directive reformats the code so that no more than one statement appears on each line. (PASMAT is described in the Utilities Guide.)

If no number is printed in the leftmost column, then that particular statement was never executed. You can sometimes detect logic errors in your program by scanning the profile output to find sections of code or perhaps entire procedures that are never executed.

A summary of the program's execution, procedure by procedure, appears at the end of the profile listing. Procedures are listed in the order they appear in your source code. Three columns of information are displayed for each procedure, as follows:

`Statements` This column lists the number of statements that appear in the definition of the procedure.

Times Called This column shows how many times each procedure is called during program execution.

Statements Executed This column has two figures. The first is the number of statements executed in the procedure. For example, a procedure that contains 10 assignment statements and is called 5 times will show 50 statements executed in the **statements executed** column. This direct relationship is valid only for very simple procedures. In most procedures and functions, loops and other control structures cause the number of "statements executed" to be much larger (or smaller) than you may expect at first glance. The second figure in this column is the percentage of statements executed in this procedure as compared to the total number of statements executed in the program. The total number of procedures and statements and the total number of statements executed are printed at the bottom of the procedure execution summary.

The following example profile from CHECKR shows that 2.6 million statements were executed. (To save space, only the Procedure Execution Summary and relevant portions of the profile listing are shown here.) The Profiler listing shows that the program spent most of its time in only a few procedures. For example, the summary shows that 21 percent of the total statements executed were in the 15-statement procedure **Check**. However, **Check** was called 71,212 times, so that percentage does not seem too far out of line. More interesting is that almost half a million statements (17.63 percent) were executed in the procedure **Initialize**. This number seems excessive because the procedure does nothing more than initialize variables and tables each time a board position is analyzed and was only called 1348 times. We may have a problem here.

```
                       PROCEDURE EXECUTION SUMMARY

Procedure name       statements   times called   statements executed

NEWNODE                  15           1390           18070   0.69%
INITIALIZE               17           1348          459668  17.63%
SCAN                     32           1348          120580   4.62%
CHECK                    15          71212          567111  21.75%
ANALYZEMOVE              40          25362          516325  19.80%
ANALYZE                  38           1348          298566  11.45%
UNPACKNODE               54           1348           60660   2.33%
PACKNODE                 23           1348           22768   0.87%
SCOREGRADIENT            15           1348          250028   9.59%
SCOREBOARD               54           1348           99228   3.80%
EVALUATEBOARD             5           1348            6740   0.26%
DISPLAYBOARD             22             41            5453   0.21%
EXTRACT                  18            715            7328   0.28%
KILL                     11           1388           13621   0.52%
PRUNE                     3            219             657   0.03%
INIT                    165              1            1575   0.06%
COMPARE                  14           4128           24768   0.95%
INSERT                   26           1843           40490   1.55%
DUMPNODE                 11              0               0   0.00%
GENMOVE                  18           1273           17822   0.68%
GENJUMP                  53             75            4389   0.17%
MOVEPIECE                12           1790           32100   1.23%
EXPAND                   17            239           20372   0.78%
POSITIONCURSOR            2              0               0   0.00%
MAKEMOVE                 55            306            7371   0.28%
DESCEND                  26            197            3592   0.14%
FULLEXPAND               45            127            6046   0.23%
READMOVE                  6              2              12   0.00%
DECODE                   12              0               0   0.00%
READFILENAME              9              0               0   0.00%
GETUSERMOVE             108              1              90   0.00%
MAIN                     91              1            2406   0.09%

There are 1032 statements in 32 procedures in this program.
   2607836 statements were executed during the profile.
```

Because we suspect a problem in the procedure **Initialize**, we examine the profile output associated with that procedure. The first column of numbers is the statement execution count. The second column is the line number of the statement in the source file. The third column of numbers is the statement number of the statement. (This statement number is the same number used by the Debugger.)

The Profiler listing for procedure Initialize is:

```
        173          procedure Initialize;
        174            var
        175              I: integer;
 1348   176     1      begin { start of Initialize }
 1348   177     2        for I := - 5 to 49 do begin
74140   178     3          Vacant[I] := false;
74140   179     4          Friend[I] := false;
74140   180     5          Enemy[I] := false;
74140   181     6          FriendKing[I] := false;
74140   182     7          EnemyKing[I] := false;
74140   183     8          Protected[I] := false;
        184                end;
 1348   185     9        Pinned := 0;
 1348   186    10        Threatened := 0;
 1348   187    11        Umobil := 0;
 1348   188    12        Denied := 0;
 1348   189    13        BlackPieces := 0;
 1348   190    14        WhitePieces := 0;
 1348   191    15        Center := 0;
 1348   192    16        MoveSystem := 0;
 1348   193    17        EnemyHasKings := false;
        194            end; { of Initialize }
```

In statements 3 through 8, a for loop is initializing several boolean arrays of the same type. Each assignment inside the loop is executed 74,140 times — a very inefficient way to initialize these arrays. Instead, we can modify the program to initialize one array, then assign that array to the other arrays to be initialized.

The effect of the modification is apparent in this new profile of the same section of code.

```
        173          procedure Initialize;
        174            var
        175              I: integer;
 1732   176     1      begin { start of Initialize }
 1732   177     2       -for I := - 5 to 49 do begin
95260   178     3          Vacant[I] := false;
        179                end;
 1732   180     4        Friend := Vacant;
 1732   181     5        Enemy := Vacant;
 1732   182     6        FriendKing := Vacant;
 1732   183     7        EnemyKing := Vacant;
 1732   184     8        Protected := Vacant;
 1732   185     9        Pinned := 0;
 1732   186    10        Threatened  :=  0;
 1732   187    11        Umobil := 0;
 1732   188    12        Denied := 0;
 1732   189    13        BlackPieces := 0;
 1732   190    14        WhitePieces := 0;
 1732   191    15        Center := 0;
 1732   192    16        MoveSystem := 0;
 1732   193    17        EnemyHasKings := false;
        194            end; { of Initialize }
```

The result is clear: Instead of six assignments, each of which is executed 74,140 times, we have one assignment executed 95,260 times. (The execution numbers differ from the sample execution summary because the CHECKR program uses random numbers to play a different game each time it is run.) Overall, the Program Execution Summary shows that the time spent in the `Initialize` procedure has dropped from 17 percent to 4 percent of the total program. By rewriting six lines, we have improved performance by 11 percent.

Further, the number of times Statement 3 is executed can be reduced by the use of a global array initialized only once at the start of the program.

Similar optimizing techniques may be applied to other parts of the program. The Procedure Execution Summary indicates where the effort can best be applied—and where it cannot. For example, the program spent 21 percent of its time in the 15-statement procedure called `Check`. The trimming of even one statement from this procedure could significantly improve performance. On the other hand, one of the larger procedures in the CHECKR program is `Genjump`, containing 53 statements. The program, however, spent much less than 1 percent of its time in this procedure. Even by eliminating this procedure completely, we would improve program performance by only a trifling amount.

Two warnings: First, a statement count is not identical to "work." Complex statements take more time to execute than simple statements and this time is not measured. Second, the percentages shown in the `statements executed` column are percentages of execution counts, not execution time. For compute-bound programs such as CHECKR, the execution percentage closely approximates the percentage of time spent in the procedures. I/O-bound programs, however, may spend much of their execution time opening files or waiting for the disk to transfer information to memory. In this case, the execution count percentages may differ significantly from the real amount of time spent in the procedures.

Pascal–2 V2.1/RSX Utilities Guide

Introduction to the Utilities Guide

The Pascal–2 utilities are a collection of programs designed to make life easier for programmers. Some of the utilities, such as the formatters, are designed to lessen the tedium in formatting programs and program documentation. Other utilities, such as the cross-reference programs, can help analyze code. Still other utilities, such as the MACRO package or the string-processing package, extend the capabilities of Pascal–2.

Each section of the Utilities Guide describes the particular utility in detail and includes examples of its use. Briefly, the Utilities Guide contains the following:

Two Program Formatters: PASMAT, a sophisticated formatter with a number of options; PB, a simple formatter designed to assist, rather than supplant, your own formatting of program text.

Two Cross-Reference Programs: XREF, which cross-references the variables in your program or words in a text file; and PROCREF, which cross-references the procedures in your program.

Dynamic String Package: STRING.PAS, a set of procedures designed to help you manipulate character strings.

MACRO Package: PASMAC, which helps to interface MACRO-11 routines with Pascal–2 programs.

Text Formatter: PROSE, which provides a number of formatting options for the production of computer-related documentation.

PASMAT: A Pascal–2 Formatter

PASMAT generates a standard format for Pascal code. PASMAT accepts standard Pascal and the language extensions in Pascal–2. PASMAT accepts full programs, external procedures, or groups of statements. A syntactically incorrect program causes PASMAT to abort and to cease formatting the output file.

PASMAT's default formatting requires no control from you. The best way to find out how the formatting works is to try it and see. In addition, PASMAT's formatting directives give you considerable control over the output format when you wish.

Overview of Capabilities

PASMAT has these capabilities:

- The program may be converted to uniform case conventions, under the control of the user.
- The program is indented to show its logical structure and to fit into a specified output line length.
- Comment delimiters are changed to braces ({}).
- If requested, the underscore character (_) is removed from identifiers for use at installations that do not support the underscore character.
- If requested, the first instance of each identifier determines the appearance of all subsequent instances of the identifier.
- All non-printing characters are removed; this feature is useful after certain editing bugs.

PASMAT handles comments, statements, and tables in the following manner:

Comments

PASMAT's rules allow you to achieve almost any effect needed in the display of comments.

- A comment standing alone on a line is left-justified to the current indention level, so that it is aligned with the statements before and after it. If it is too long to fit with this alignment, it is right-justified.
- A comment that begins a line and continues to another line is passed to the output unaltered, indention unchanged. This type of comment is assumed to contain text formatted by the author, so it is not formatted.
- If a comment covered by one of the above rules does not fit within the defined output line length, the output line is extended as necessary to accommodate the comment. Once formatting is complete,

a message to the terminal gives the number of times the width is exceeded and the output line number of the first occurrence.

- A comment embedded within a line is formatted with the rest of the code on that line. Breaks between words within a comment may be changed to achieve proper formatting, so nothing that has a fixed format should be used in such a comment. If a comment cannot be properly spaced so that the line fits within the output length, that line is extended as necessary. Once formatting is complete, a message to the terminal gives the number of times the width is exceeded and the output line number of the first occurrence. If no code follows a comment in the input line, then no code is placed after the comment in the output line.

Statement Bunching

The normal formatting rule for a `case` statement places the selected statements on a separate line from the `case` labels. The `B` directive (see "Formatting Directives") tells the formatter to place these statements on the same line as the `case` labels if the statements fit.

Similarly, the rules for `if-then-else`, `for`, `while`, and `with` place the controlled statements on separate lines. The `B` directive tells the formatter to place the controlled statement on the same line as the statement header if the statement fits.

Tables

Many Pascal programs contain lists of initialization statements or constant declarations that are logically a single action or declaration. You may want these to be fit into as few lines as possible. The `S` directive (see below) allows this. If this is used, logical tab stops are set up on the line, and successive statements or constant declarations are aligned to these tab stops instead of beginning on new lines.

At least one blank is always placed between statements or comment declarations, so if tab stops are set up at every character location, statements are packed on a line.

Structured statements, which normally format on more than one line, are not affected by this directive.

Using PASMAT

Invoke PASMAT with the following command:

>PASMAT *output-file*=*input-file*/options="*directives*"

where

input-file Is the Pascal source file being reformatted. PASMAT accepts only one input file. The default file extension for both input and output is .PAS.

output-file Is the reformatted Pascal source file. If *output-file* is omitted, the output file receives the same file name and extension as the input file and becomes the latest version of that file.

options="*directives*" Are the settings for formatting directives. The options switch is optional. It may be abbreviated to o and may be placed anywhere on the command line. Though the '=' is shown as the switch separator, a colon (:) may also be used between the options switch and the directives. When specified on the command line, directives must be placed in quotes as shown. The *directives* field are scanned as though the directives were in a Pascal comment at the start of the source program.

If the command, PASMAT, is entered on a line by itself, PASMAT prompts for file names. At the PMT> prompt, enter the remainder of the command line.

Formatting Directives

Formatting directives may be specified either by an `options` switch on the command line or by a special form of the Pascal comment structure.

Formatting directives are of two kinds: switches that turn *on* with the plus sign (+) and *off* with the negative sign (-) (e.g., `R+` and `L-`); or numeric directives of the form `T=5`. Multiple directives are separated by commas (e.g., `R+,L-`). Blanks are not allowed within a directive. Case is ignored: `R+` is the same as `r+` in a directive.

By definition (and by default), certain directives override other directives, such as the L directive overriding the U and R directives. Therefore, when turning on a directive, you must turn off any directive that overrides it. For example, suppose you want all Pascal reserved words in upper case. In addition to setting `R+`, which specifies upper case, you must also turn off the L directive with `L-`. See the second example under "PASMAT Examples."

The following example shows a program named PROG.PAS being formatted with a command-line directive that sets the switch B on, R off and the numeric directives O to 72 and T to 5.

```
>PASMAT PROG/OPTIONS="B+,O=72,T=5,R-"
```

If used in the program text as part of an embedded Pascal comment, format directives are placed within square brackets that, along with any other comments, are placed within the standard Pascal comment braces. A compiler directive (e.g., `$nomain`), if present, must begin any comment containing a PASMAT directive. In this case, the PASMAT directive may come before or after any other text:

{`$`*compiler-directives text* [*directives*] *text*}

If no compiler directive is present, the PASMAT directive must begin the comment:

{[*directives*] *text*}

The following embedded directive has the same effect as the command-line directive shown above.

```
{[b+,o=72,t=5,r-]}
```

The PASMAT formatting directives are:

A (Default `A-`) Adjusts each identifier so that the first instance of the identifier determines the appearance of all subsequent instances of the identifier. This facility standardizes the use of upper-case and lower-case characters and the underscore character (_) in program text. This directive overrides the U directive.

B (Default B-) Specifies that the statements following a **then**, or **else**, **for**, **with** or **while** is placed on the same line if they fit. The statement following a **case** label is placed on the same line if it fits. The result is a shorter output, which may be easier to read but which also may be harder to correct.

C (Default C-) Converts leading blanks to tabs on output.

F (Default F+) Turns formatting on and off. This directive goes into effect immediately after the comment in which it is placed and can save carefully hand-formatted portions of a program.

K (Default K-) Converts a non-standard **else** clause in a **case** statement to **otherwise** as used in Pascal–2.

L (Default L+) Specifies that the case of identifiers and reserved words be a literal copy of the input. This directive overrides the U and R directives and is disabled by the P+ directive.

M (Default M+) Converts all alternate symbol representations to the standard form. Otherwise, all symbols are left as they are in the text. The nonstandard comment brackets /* ... */ are always converted, either to braces or, in the case of M-, to (* ... *).

N (Default N-) Inserts no new lines into the output unless they are required to make the lines fit. This directive just indents the source, keeping the line structure set up by the user. If a line exceeds the output length, it is broken at the best place available, but the results may not be what you want. Look things over carefully after using this option.

O (Numeric directive, default O=78) Specifies the width of the output line. The maximum value allowed is 132 characters. If a particular token does not fit in the width specified, the line is lengthened accordingly, and a message at the end of the formatting gives the number of times the width is exceeded and the output line number of the first occurrence.

P (Default P-) Sets "portability mode" formatting, which removes underscore characters (_) from identifiers. The first letter of each identifier, and the first letter following each underscore, is made upper case, while the remaining characters are in lower case. This directive overrides the L and U directives. The R directive sets the case of reserved words.

Warning: Pascal–2 considers underscore characters significant: User_DoesThis is one identifier and UserDoes_This is another. Take care when using this directive that you do not make two different identifiers the same: UserDoesThis and UserDoesThis.

R (Default R-) Specifies that all reserved words are in upper case. With this off, reserved words are in lower case. The L directive overrides the R directive. When using R+ you must also use L- to turn off the overriding directive. See the second example under "PASMAT Examples."

S (Numeric directive, default S=1) Specifies the number of statements per line. The space from the current indention level to the end of the line is divided into even pieces, and successive statements are put on the boundaries of successive pieces. A statement may take more than one piece, in which case the next statement again goes on the boundary of the next piece. This is similar to the tabbing of a typewriter.

Any statement requiring more than one line is not affected, but may cause unexpected results on following statements. This directive only affects the constant declaration and statement portions of the program and is intended for use in initializing tables. The default value of 1 provides normal formatting.

T (Numeric directive, default T=2) Specifies the amount to "tab" for each indention level. Statements that continue on successive lines are additionally indented by half the value of T.

U (Default U-) U+ specifies that identifiers are converted to upper case; U- specifies that they are converted to lower case. The L, P and A directives override this directive. When using U+ you must also use L- to turn off the L directive. Also, make sure the P directive is off (P-, the default).

Limitations and Errors

PASMAT is limited in these ways:

- The maximum input line length is 132 characters.
- The maximum output line length is 132 characters.
- Only syntactically correct statements are formatted. A syntax error in the code causes the formatting to abort. An error message gives the input line number on which the error is detected. The error checking is not perfect, and successful formatting is no guarantee that the program compiles.
- The number of indention levels handled by PASMAT is limited; PASMAT aborts if this number is exceeded — a rare circumstance.
- If a comment requires more than the maximum output length (132) to meet the rules given, processing is aborted. This situation should be even rarer than indention-level problems.
- When it aborts, PASMAT attempts to copy the rest of the file. You should, however, recover a copy of the source file and inspect the PASMAT-generated copy carefully; we cannot guarantee that PASMAT recovers all the text for every error condition.

PASMAT Examples

To illustrate the workings of various PASMAT options, the following example shows how a sample program appears after formatting with two different sets of options.

Example: EFACT

```
program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                  }
var E, series_term: real; N: integer;
begin
{ set initial conditions }
E := 1.0; N := 1; SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
repeat
E := E + seriesterm;
{ compute next term of series }
N := N + 1; seriesterm := seriesterm / N;
until E = (E + SeriesTerm);
writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
end.
```

First we reformat the program using the standard indention of text and comments. We use the output directive on the command line to specify the width of the output line, and we specify a short line width to illustrate the right-justification of long comments.

The program is formatted with the commands:

```
>PASMAT EFACT/OPTIONS="O=66"
```

Program text after formatting:

Example: EFACT - Output

```
program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                  }

  var
    E, series_term: real;
    N: integer;

  begin
    { set initial conditions }
    E := 1.0;
    N := 1;
    SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
    repeat
      E := E + seriesterm;
      { compute next term of series }
      N := N + 1;
      seriesterm := seriesterm / N;
    until E = (E + SeriesTerm);
    writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
  end.
```

The second example illustrates embedded PASMAT commands. We have altered the original program by inserting the text {[A+,L-,R+]} before the first line. The directive A+ changes each identifier to match the appearance of the first use of that identifier. (Notice the variant forms of series_term and E in the original program.) The directives L- and R+ together turn off the literal reproduction of the reserved words and make them upper case. The program is formatted with the commands:

```
>PASMAT EFACT
```

Program text after formatting:

Example: EFACT after formatting

```
{[A+,L-,R+]}
PROGRAM Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                  }

  VAR
    E, series_term: real;
    N: integer;

  BEGIN
    { set initial conditions }
    E := 1.0;
    N := 1;
    series_term := 1.0;
    { loop to approximate E; quit when the series sum stops changing }
    REPEAT
      E := E + series_term;
      { compute next term of series }
      N := N + 1;
      series_term := series_term / N;
    UNTIL E = (E + series_term);
    writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
  END.
```

XREF: A Pascal-2 Cross-Reference Lister

XREF produces a cross-reference listing of the identifiers in a Pascal program. XREF is helpful when debugging new programs or when modifying existing ones. The output shows the use of each identifier in the program, which is beneficial when you're working with medium to large programs.

Each identifier is listed, along with an entry for each reference to that identifier. Each entry consists of the line on which the reference occurs, plus an indication of whether the reference is a declaration or assignment.

Using XREF

You invoke XREF with the following command:

>XREF ***output-file=input-file/switches***

where

input-file Is the Pascal source file being cross-referenced. The input file has a default extension of .PAS. XREF accepts only one input file.

output-file Is the cross-reference file. The output file has a default extension of .CRF. *Output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the input file, and having the default extension, is placed in the default directory.

switches Is one or both of the following command-line switches.

The `list` switch generates a listing of the input file before the cross-reference. This listing includes line numbers and a flag character (c) indicating multiple line comments and strings. The flag character makes it easier to locate certain bugs that cannot be easily diagnosed by the compiler.

The `width:`*num* switch specifies the page width for the cross-reference listing, where *num* is the number of characters across. The default is 132.

The switches may be abbreviated to one letter. Multiple switches are separated by a slash '/'.

Limitations

The XREF program has two limitations on the size of the programs it can handle.

- An internal limit exists for the number of distinct identifiers allowed. You may change this number in the XREF source file and recompile the program.
- The total number of references is limited by the amount of dynamic storage available.

The XREF program does not perform a complete syntax analysis of the program, and it may not flag all declarations or assignments.

This example shows the cross-referencing of the program EFACT to produce the output file EFACT.CRF.

```
>XREF EFACT/LIST/WIDTH:66
```

Example: XREF Listing

```
         1 program Efact(output);
         2 { Compute an approximation for E from its Taylor series.
c        3    The Nth term in the series is 1/(N!).
c        4 }
         5
         6   var
         7     E, SeriesTerm: real;
         8     N: integer;
         9
        10   begin
        11     { set initial conditions }
        12     E := 1.0;
        13     N := 1;
        14     SeriesTerm := 1.0;
        15     repeat  { loop to approximate E; quit when sum stops changing }
        16       E := E + SeriesTerm;
        17       N := N + 1;
        18       SeriesTerm := SeriesTerm / N;
        19     until E = (E + SeriesTerm);
        20     writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
        21   end.


Cross reference:  * indicates definition, = indicates assignment

-E-
E                  7*    12=    16=    16     19     19     20
EFACT              1*

-I-
INTEGER            8

-N-
N                  8*    13=    17=    17     18     20



-O-
OUTPUT             1*

-R-
REAL               7

-S-
SERIESTERM         7*    14=    16     18=    18     19

-W-
WRITELN            20

end xref   8 identifiers    24 total references
```

PROCREF: Pascal–2 Procedural Cross-Reference Lister

PROCREF, based on a procedural cross-reference program published by Arthur Sale in *Pascal News* (Number 17, March 1980), is designed to help programmers sort through the procedures in medium to large Pascal programs. The program has been modified to allow the use of multiple input files and `%include` directives and to provide "called by" data in the listing.

PROCREF provides a quick overview of the procedural organization of a program, which is beneficial when you're working with medium to large programs. The PROCREF utility reads the text of a Pascal program to produce a compact listing of the procedure headings and an alphabetized list of procedures with usage information. PROCREF processes `%include` directives in the same way as the Pascal–2 compiler, so that all parts of a compilation can be analyzed.

The procedure listing includes each procedure heading, along with its location in the input file. Procedure headings are indented to show lexical level. No attempt is made to fit the procedure headings into a limited line width.

The cross-reference listing places procedures in alphabetical order. For each procedure the listing includes:

- The file and line number where its heading starts.
- The file and line number where its body starts, unless it is external or is a formal procedure parameter and has no body. In such a case, the note `external` or `formal` is printed.
- If the procedure is declared `forward` or is externally defined, the listing contains the file and line number where the procedure heading stub starts.
- A list of all procedures immediately called by this procedure. These are listed in the order in which they occur in the text. A procedure is listed only once, even if it is called more than once.
- A list of all procedures that call this procedure. Again, the list is in textual order and only one reference is shown per procedure.

Only the first sixteen characters of a procedure name appear in the cross-reference listing. Those characters are written exactly as they appear in the program text.

Using PROCREF

You invoke PROCREF with the following command:

```
>PROCREF output-file=input-files/width:num
```

where

input-files Are the Pascal source files being cross-referenced. The input files have a default extension of .PAS. Multiple input files, if specified, are separated by commas. Multiple files are concatenated.

output-file Is the cross-reference file. The output file has a default extension of .PRF. The *output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the last input file, and having the default extension, is placed in the default directory.

`width`:*num* Specifies the page width for the cross-reference listing, where *num* is the number of characters across the page. The default is 80 characters. The `width` switch is optional and may be abbreviated to one letter.

Limitations

The PROCREF program does not completely analyze the syntax of the program being processed. PROCREF errs in one case: If a field identifier in a record has the same name as a procedure, and if that field is referenced without a preceding record variable name, as in a `with` statement, the field identifier is treated as a reference to the procedure.

Let's assume that we wish to generate a procedure cross-reference for the following program, LVSPOOL.PAS.

Example: Program Listing For LVSPOOL.PAS

```
Pascal-2 RSX V2.1E   9-Feb-88   7:03 AM     Site #1-1     Page 1-1 Oregon Software, 6915
SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202 LVSPOOL/LIST
    1      program LVSpool(input, output);
    2        procedure ScanLV; external;
    3        procedure ReadFontInfo(i: integer; j: integer); forward;
    4
    5        procedure LoadFonts;
    6          procedure GetByte;
    7          begin
    8          end;
    9        begin
   10          GetByte;
   11          ReadFontInfo(1, 2);
   12        end;
   13
   14        procedure ReadFontInfo;
   15        begin
   16          LoadFonts;
   17        end;
   18
   19        procedure ShowPage;
   20        begin
   21          ScanLV;
   22        end;
   23
   24      begin            main program
   25        ReadFontInfo(0,1);
   26        ShowPage;
   27      end.


*** No lines with errors detected ***
```

We cross-reference the procedures as follows.

```
>PROCREF LVSPOOL=LVSPOOL/W:72
```

The `W:72` requests that the cross-reference listing not exceed 72 characters in width so that the result may be printed on a terminal.

The result, placed in the file LVSPOOL.PRF, consists of:

Example: Output From PROCREF (LVSPOOL.PRF)

```
Procedural Cross-Referencer - Version 3.0
LVSPOOL/W:72

 Line Program/procedure/function heading
-----------------------------------------

LVSPOOL.PAS:

    1  program LVSpool(input, output);
    2      procedure ScanLV; external;
    3      procedure ReadFontInfo(i: integer; j: integer); forward;
    5      procedure LoadFonts;
    6          procedure GetByte;
   14      procedure ReadFontInfo;
   19      procedure ShowPage;

Procedural Cross-Referencer - Version 3.0
LVSPOOL/W:72

Cross Reference Listing

GetByte              Head: LVSPOOL.PAS, 6  Body: LVSPOOL.PAS, 7
      Called by      LoadFonts

LoadFonts            Head: LVSPOOL.PAS, 5  Body: LVSPOOL.PAS, 9
      Calls          GetByte            ReadFontInfo
      Called by      ReadFontInfo

LVSpool              Head: LVSPOOL.PAS, 1  Body: LVSPOOL.PAS, 24
      Calls          ReadFontInfo       ShowPage

ReadFontInfo         Head: LVSPOOL.PAS, 3  Body: LVSPOOL.PAS, 15
                     Forward, header stub: LVSPOOL.PAS, 14
      Calls          LoadFonts
      Called by      LoadFonts          LVSpool

ScanLV               Head: LVSPOOL.PAS, 2  external
      Called by      ShowPage

ShowPage             Head: LVSPOOL.PAS, 19  Body: LVSPOOL.PAS, 20
      Calls          ScanLV
      Called by      LVSpool
```

Dynamic String Package

The Pascal standard implements character strings in two ways: as a sequence of two or more characters between single-quote marks (a literal string); or as a **packed array of char** (a variable string). However, the standard does not provide adequate facilities for manipulating character strings and only allows assignments of one string to another string and comparisons of two strings of equal length.

Pascal–2's Dynamic String Package extends the meager string-handling capabilities of standard Pascal, providing the ability to perform sophisticated operations on strings of varying lengths. The string package, STRING, is a collection of string-processing procedures and functions that allows Pascal programs to read and write strings, concatenate two strings, search one string for another, insert one string into another and delete one string from another, assign the value of one string to another string and other string operations. The string package is written in standard Pascal to take advantage of conformant array parameters, which facilitate the passing of variable-length arrays (strings), and to provide portability to other Pascal implementations.

To use the string package, declare string variables as packed arrays of characters with a lower bound of 0 and an upper bound equal to the maximum length for that particular string, as shown:

```
var
  string-name: packed array [0..max-len] of char;
```

where *string-name* is the identifier associated with the string variable and *max-len* is the maximum length of the string in bytes. The actual length of the string is stored in element 0. The characters making up the string are stored starting at element 1. *Max-len* must be greater than 0 and no larger than 255.

The maximum length of a string may be different for each string, depending on the intended use ~~of the~~ string. The string package's use of conformant array parameters makes this possible. Examples:

```
var
  NameString: packed array [0..25] of char;
  SiteNo: packed array [0..7] of char;
  LineOfInput: packed array [0..80] of char;
```

As an alternative, these routines also accept parameters of type **packed array [1..*max-len*] of char**, where *max-len* is the actual length of the string. Literal strings are of this type. This means you may pass a literal string to any of these procedures as long as the formal parameter is not a **var** parameter.

STRING may be included in program source files in one of two ways: in the program code, use the `%include` compiler directive; or on the command line, concatenate STRING.PAS with the rest of the source files making up the program ("source concatenation"). We recommend the use of the `%include` directive:

```
%include '[1,1]string';
```

However, if you concatenate the string package with the source file, the command to compile program PROG is:

```
>PAS STRING,PROG
```

Source concatenation may be used only if the main program does not contain a **program** statement; otherwise, compilation errors result. Refer to "Multiple Source Files" in the Programmer Reference for more information on the `%include` directive.

The Procedures and Functions

In the definitions below, *string* and *target* represent string variables similar to the previous examples. *File* must be a variable of type `text`. *Start* and *span*, of type `integer`, represent character positions and character ranges, respectively. *Max-len* is the upper boundary, or maximum length, of the array. *Char* may be a variable of type `char` or a literal string of one character.

The string package contains these procedures and functions:

`Len(`*string*`)` An integer function, returns the actual length of *string*. *String* may be a literal string.

`Clear(`*string*`)` Initializes *string* to empty.

`ReadString(`*file*`, `*string*`)` Reads *string* from *file*. The string is terminated when `eoln(`*file*`)` becomes `true`, and a `readln(`*file*`)` is performed. Overflow results in truncation to *max-len* characters.

`WriteString(`*file*`, `*string*`)` Writes *string* to *file*. This procedure does not accept literal strings as parameters. Use `writeln` to terminate a written string manually.

`LeftString(`*target*`, `*string*`, `*First*`)` Returns a string and the location of the first character to the right of the string.

`RightString(`*target*`, `*string*`, `*Last*`)` Returns a string and the location of the last character to the left of the string.

`Concatenate(`*target*`, `*string*`)` Appends *string* to *target*. The resulting value is *target*. *String* may be a literal string. Overflow results in truncation to *max-len* characters.

`Search(`*string*`, `*target*`, `*start*`)` Searches *string* for the first occurrence of *target* to the right of position *start* (characters are numbered beginning with 1). The `Search` function returns the position of the first character in the matching substring, or the value zero if *target* does not appear in *string*. *String* and *target* may be literal strings.

`Insert(`*target*`, `*string*`, `*start*`)` Inserts *string* into *target* at position *start*. Characters are shifted to the right as necessary. Overflow produces a truncated *target* of *max-len* characters. The insertion is skipped if the *start* position causes a non-contiguous string. *String* may be a literal string.

`Assign(`*target*`, `*string*`)` Assigns *string* to *target*. This procedure is especially useful for assigning a literal *string* to a variable string (*target*). To assign one character to a variable string, use the `Asschar` procedure, below.

`Asschar(`*target*`, `*char*`)` Assigns *char* to *target*. *Char* may be a literal character or a variable name. This procedure is more efficient than procedure `Assign` for the creation of one-character strings. (With `Assign`, a one-character string must first be created as input to `Assign`, which then assigns the character to a variable string.)

`Equal(`*target*`,` *string*`)` Determines whether *target* is element-for-element identical to *string*. This boolean function returns a `true` value if the two strings are equal, `false` if the two strings are different. *Target* and *string* may be literal strings.

The *start* and *span* parameters in the `Delstring` and `Substring` procedures define a substring beginning at position *start* (between characters *start*-1 and *start*) with a length of `abs(`*span*`)`. If *span* is positive, the substring is to the right of *start*; if negative, the substring is to the left.

`Delstring(`*string*`,` *start*`,` *span*`)` Deletes the substring defined by *start*, *span* from *string*. (In previous versions of Pascal-2, this procedure was named `delete`.)

`Substring(`*target*`,` *string*`,` *start*`,` *span*`)` The substring of *string* defined by *start*, *span* is assigned to *target*. *String* may be a literal string.

The sample program PDLIST.PAS demonstrates the use of the string package. The program reads a PROSE input file (.PRS extension) of text-processing commands, or "directives," searching for all directives used in the file. (PROSE is described later in this guide.) As each directive is encountered, PDLIST prints the directive and its location within the file for future reference.

The first character of a PROSE directive is called the "escape" character. If the first character of a line of input is an escape character, at least one directive follows. PDLIST.PAS uses the `Readstring` procedure to read a line of text as a string, then calls `Search` repeatedly to find each occurrence of the escape character on the current line. When an escape character is found, the procedure `GetDirective` is called to get the next directive. For each directive, the program builds a line of output (also a string) using the `Assign` and `Concatenate` procedures, and uses `Writestring` to write the line to a .DTV (directive) file. In this example, the escape character is a period, which is the PROSE default.

PDLIST.PAS, including procedure `GetDirective`, is provided in full on the following pages. Sample execution follows the listing.

Example: Use of L1String (PDLIST.POS)

```
program DirectiveList;
  %include 'string'; ——————————— include the string package

  const
    LineLength = 150;    { PROSE default input line length }

  var
    Line: packed array [0..LineLength] of char;  { string for output line }
    Outline: packed array [0..50] of char;       { string for input line }
    Directive: packed array [0..10] of char;     { string for directive }
    Name: packed array [1..80] of char;          { input file name }
    Escape: packed array [0..1] of char;         { string for escape character }
    Linenum, Index: integer;
    Prosefile, Directivefile: text;
    DirectiveFound: boolean;       { directive found? }
    NoneYet: boolean;              { looking for first directive on line }
    Letters: set of char;          { Characters making up a directive }

procedure GetDirective(I: integer);
  var
    Ch: char;

  begin  { GetDirective }
    Clear(Directive); ——————————— defined in STRING
    while I <= Len(Line) do begin    { not end of line yet }
      Ch := Line[I];  I := I + 1;
      if Ch in Letters then begin     { get next char of directive }
        Directive[0] := succ(Directive[0]);
        Directive[Len(Directive)] := Ch
        end
      else I := Len(Line) + 1;
      end;
    DirectiveFound := Len(Directive) > 0
  end;    { end GetDirective }
  begin    { DirectiveList }
    write('PROSE FILE: ');
    readln(Name);
    reset(Prosefile, Name, '.prs');
    rewrite(Directivefile, '.dtv', Name);
    Asschar(Escape, '.');——————————— defined in STRING
    Linenum := 0;
    Letters := ['A'..'Z', 'a'..'z'];      { Any others end directive }
    while not eof(Prosefile) do begin
      ReadString(Prosefile, Line);——————— defined in STRING
      Linenum := Linenum + 1;
      if Line[1] = Escape[1] then begin   { first character is an escape }
        Index := 0;  NoneYet := true;
        repeat       { find all occurrences of escape characters }
          Index := search(Line, Escape, Index + 1);——— defined in STRING
          if Index <> 0 then begin
            GetDirective(Index + 1);       { get the next directive }
            if DirectiveFound then begin
              Assign(Outline, Escape);————— defined in STRING
              Concatenate(Outline, Directive);———— defined in STRING
              if NoneYet then begin
                write(Directivefile, ' Line ', Linenum: 4,'     ');
                NoneYet := false
                end
              else write(Directivefile, '              ');
              WriteString(Directivefile, Outline);——— defined in STRING
              writeln(Directivefile);
              end;
          end;
        until Index = 0;
        end;
    end;
end.  { DirectiveList }
```

For this illustration, PDLIST reads the PROSE input file presented in Appendix A of "PROSE: A Text Formatter," Example 2. The name of the input file is PEXAM2.PRS.

Compile, link, and run PDLIST as usual. The program prompts for the input file:

```
PROSE FILE: PEXAM2
```

Output, the list of directives used in the file, is written to PEXAM2.DTV, which looks like this:

```
Line    1   .COMMENT
Line    2   .INPUT
Line    3   .OPTION
Line    4   .FORM
Line    5   .MARGIN
Line    6   .PARAGRAPH
Line   21   .OPT
Line   22   .MAR
Line   23   .PARAGRAPH
Line   28   .OPT
            .MAR
            .PAR
```

MACRO-11 Procedures With Pascal–2

Although most programs can be written within the Pascal–2 language, applications involving interface to the operating system require the use of MACRO-11 assembly language code. A set of macros provided with the Pascal–2 system makes this interface easy. You can code a set of macro calls that look much like a Pascal procedure declaration, and the PASMAC macro package will assign addresses to the parameters and generate procedure entry and exit code.

Design of MACRO-11 Procedures

Follow these general rules in deciding what to put in a MACRO-11 procedure:

- Do the absolute minimum in MACRO-11. If you must use MACRO-11 code to use a system service, process the result in Pascal–2 code. (This is not always possible, since some operating systems require very low-level manipulations.)
- Isolate a common function and make the procedure handle the most general case of that function.
- Pass all data to and from the procedure as parameters. Global references from MACRO-11 are not recommended for these reasons: the address is hard to find; if the Pascal program changes, the MACRO program will have to be changed; and global references cannot be checked for type compatibility. This guide does not describe ways to make global references.

Once you have decided on the contents of the procedure, define the calling sequence as a Pascal external procedure. Then write a functional description of the procedure. Then actually write the procedure. These documents will be your implementation guide.

When you have the external definition, use the PASMAC macro package described below to define parameters and local variables. As long as the stack is not changed within the procedure, these macros can access parameters or local variables directly. For this reason, you should probably store local temporary values in the local variables rather than pushing them on the stack. If thoroughly familiar with writing MACRO-11 code, you can use the stack, but make sure you understand the Pascal–2 run-time structure, described in the Programmer Reference.

The PASMAC Macro Package

The PASMAC macro package is provided to simplify the writing of MACRO-11 procedures to interface with Pascal-2. Using this package, you can declare procedures, parameters, and variables, and you can easily refer to these items within the procedure.

The package consists of the following macros:

The PASMAC package also contains assembly language routines that define the macros:

Name	Arguments	Function
proc	*procname*	Begins the declaration for the procedure *procname*.
func	*funcname* *result restype*	Begins the declaration for the function *funcname*. The returned value is assigned to *result* of type *restype*.
param	*parmname* *parmtype*	Declares a parameter named *parmname* of type *parmtype*.
var	*varname vartype*	Declares a local variable named *varname* of type *var-type*.
save	<*reg0*, ... ,*regn*>	Specifies general registers to save on procedure entry.
rsave	<*ac0*, ... ,*acn*>	Specify floating accumulators to save on procedure entry.
begin		Begins the actual procedure code. This macro generates code to push the variables on the stack and to save registers.
endpr		Ends the code for this procedure, restores registers, pops variables and parameters from the stack, and returns to the calling location.

The following example demonstrates how these macros may be used in a procedure definition. Note the correspondence between the Pascal-2 code and the MACRO-11 code.

Pascal-2 procedure definition:

```
procedure Exampl(Inp1: integer;      { first value parameter }
                 Inp2: real;         { second value parameter }
                 var Outp: integer   { variable parameter });

var
  Var1: integer;                     { first local variable }
  Arr1: array [1..3] of integer;     { second local var }

begin                                { begin body of procedure }
:
 ----------------------------------- procedure code
end;                                 { end of procedure }
```

The corresponding MACRO-11 code:

```
proc    exampl              ; declare the procedure
param   inp1,integer        ; first value parameter
param   inp2,real           ; second value parameter
param   outp,address        ; variable parameter

var     var1,integer;       ; first local variable
var     arr1,3*integer      ; second local variable

save    <r0,r1>             ; registers being used
rsave   <ac0,ac1>           ; floating accum being used

begin                       ; begin body of code
:
 ------------------------- procedure code
endpr                       ; reset everything and return
```

Using PASMAC

The macros described in the following sections are included in the file PASMAC.MAC, which also includes definitions of standard data types. It is assumed that this file will be assembled as a header to any MACRO-11 code. This would normally be done with a command line similar to:

```
>MAC EXTPRO,EXTPRO=PASMAC,EXTPRO
```

The result of this assembly is an object file (.OBJ) that is task-built in the same way as any other external module.

MACRO-11 modules assemble with the PASMAC package are referenced from Pascal via the **external** directive instead of the **nonpascal** directive, because PASMAC simulates the Pascal calling sequence.

(MACRO-11 routines assembled without the PASMAC package can be referenced via the nonpascal directive.) For example:

```
procedure ExtProc(Parm1: integer);
  external;
```

For details, see "External Modules" in the Programmer Reference.

The example command line above also generates a listing file (.LST). Listing of the PASMAC file is disabled with a .NLIST directive at the start of the file. A compensating .LIST directive is placed at the end of the file, so a program listing is not affected. Defining the tag $LIST anywhere in your code will enable listing of the PASMAC file.

The macros depend on the existence of a uniform radix throughout the declaration of a single procedure. This radix may be octal or decimal, but it must not be changed within a procedure declaration. Also, the macros use labels of the form Q$xxx and macros of the form $Pxxx for storing state data. Avoid such forms in your own code.

Procedure Definition Macros

The PASMAC procedure definition macros must be used in the order:

Macro	Usage
`proc/func`	Exactly one of these is required
`param`	As many as required (or none)
`var`	As many as required (or none)
`save/rsave`	Either or both as needed
`begin`	Required
⋮	User code
`endpr`	Required

A MACRO-11 error is detected if the macro calls are not made in the required order.

Above references to parameter and variable "types" assume that "type" identifiers are equivalent to the length of a value of that type. For example, the identifier `integer` has the value of 2, the identifier `real` has the value of 4, and a disk buffer may have the value of 512. The PASMAC package defines some standard types. See "Type Definitions" below.

Parameter, variable and function result names are set to offsets relative to the value of the stack pointer at the end of the **begin** macro. This takes into account local variables allocated on the stack, plus the space used for register saving. You must take into account any additional values you pushonto the stack.

Examples:

```
param    param1,integer     ; defines param1
:

mov      param1(sp),r0      ; use param1
```

The 'Proc' Macro

The proc macro, used to begin the definition of a procedure, specifies the name to be used and initializes the symbols that store data about the procedure. This macro must be the first macro used in a procedure declaration.

The calling sequence is:

`proc` *procname*[, `check=1`]

where

procname is the name to be used to call the procedure. Only the first six characters of this name are significant.

check is an optional parameter specifying stack overflow checking. A non-zero value (default) requests a stack overflow check. This check is free (and always done) if more than three registers are saved, and costs two words in the procedure entry otherwise. The time for the check is very small, so disabling it is not recommended.

Examples:

```
proc    p,check=0
  or:
proc    p,0
```

Begins the declaration of a procedure with the external name p and no stack overflow checking.

```
proc    savetime
```

Begins the declaration of a procedure with the external name saveti and stack checking enabled.

The 'Func' Macro

The func macro, similar in function to the proc macro, also allows you to specify a name and type for the returned value. In Pascal, the returned value is specified by assignment to the function name. In MACRO-11, this assignment is not possible, since the function name is used for the procedure entry and cannot also point to the appropriate place on the stack. Any value assigned to the result name defined in the func macro at exit from the function is returned as the function value.

The calling sequence is:

func   *funcname, resname, restype*[, check=1]

where

funcname is the name to be used to call the function. Only the first six characters are significant.

resname is the name to be used to reference the returned value. Any value assigned to this location during execution is returned to the calling program upon exit from the procedure.

restype is the length of the result value. This is not used in the current implementation of the macros, but is included for documentation and possible future use.

check is an optional parameter that enables stack checking if non-zero. See the description under the proc macro.

Example:

```
func     curtime,tval,real
```

Begins the declaration of a function with the external name `curtim` and stack overflow checking enabled. The result location will be named `tval`, of type real. Here `real` is assumed to have the value 4, which is the length of a single-precision real value.

The 'Param' Macro

The `param` macro specifies parameters to the current procedure or function. Each parameter has one `param` macro, in the order declared in the Pascal procedure declaration. In the Pascal-2 calling sequence, parameters are pushed onto the stack in the order in which they are declared, so the first parameter is at a higher address than the last parameter. Value parameters have the actual value pushed, and variable parameters have the address of the variable pushed. When these parameters are declared, the parameter name is set equal to the offset of that parameter relative to the stack pointer (`sp`) after the `begin` macro has been called. This value may be used to access the parameter location relative to the stack pointer.

The calling sequence is:

`param`   *paramname*, *paramtype*

where

paramname is the name to be used for accessing the parameter. Within the body of the procedure, if the stack pointer (`sp`) has not changed since the `begin` macro, value parameters can be referred to by *paramname*`(sp)`, and variable parameters can be referred to as `@`*paramname*`(sp)`.

paramtype is the data type used to determine the space on the stack used by this parameter.

Examples:

```
param    input,integer       ; input: integer
param    result,address      ; var result: integer
```

These macros define two parameters. The first is a value parameter with the name `input` of type `integer` and is referred to in the body of the procedure as `input(sp)`. The second is a variable parameter with the name `result` of type `integer`. Note that the type is defined only in the comment; the actual value pushed on the stack is of type `address`. Within the body of the procedure this is `@result(sp)`.

The 'Var' Macro

The var macro, similar to the param macro, defines a local variable to be allocated on the stack upon procedure entry. The space for these variables is allocated automatically by the begin macro, but is not initialized. Such variables are referenced relative to the stack pointer (sp).

The calling sequence is:

```
var     varname, vartype
```

where

varname is the name to be used for accessing the variable. Within the body of the procedure, if the stack pointer (sp) has not been modified since the begin macro, variables can be referred to by *varname*(sp).

vartype is the data type used to determine the space to be allocated for this variable.

Example:

```
var     temp,integer        ; temp: integer;
var     name,10*char        ; name: array [1..10] of char;
```

The example defines two local variables. The space for these variables will be pushed onto the stack by the begin macro. The variable temp has two bytes allocated and is referred to as temp(sp). The variable name has ten bytes allocated and is referred to as name(sp).

The 'Save' Macro

The save macro specifies the general registers to be saved on procedure entry. The Pascal–2 calling conventions require a procedure to save and restore all registers used within a procedure, so any registers altered within the procedure should be listed here. If more than three registers are to be saved, a routine from the Pascal support library is used to save the registers. The stack pointer and program counter (sp and pc) cannot be saved.

The calling sequence is:

```
save    <reg1, ..., regn>
```

where <*reg1*, ..., *regn*> is a list of registers to be saved, enclosed in angle brackets (<>) and separated by commas. These registers will be saved on entry and restored on exit. The registers sp and pc cannot be saved, as they are modified by the action of saving them.

Examples:

```
save    <r0,r1>
```

Save registers R0 and R1 and restore them on exit. The code generated uses explicit mov instructions to do this.

```
save    <r0,r1,r2,r3,r4,r5>
```

Save and restore all available registers. Support routines will be used.

The 'Rsave' Macro

The **rsave** macro is useful only for machines with the Floating Point Processor (FPP) hardware option and serves the same function as **save** except for the floating-point accumulators. You are required to specify the FPP mode, either single or double (default is single). Since the accumulators **AC4** and **AC5** cannot be moved directly to memory, they may not be used unless one of the accumulators **AC0** to **AC3** is also used. Of course, you cannot get data into **AC4** or **AC5** without using one of the lower accumulators, so you should not have any problems meeting this requirement.

The calling sequence is:

```
rsave    <accum1, ..., accumn>[, double=0]
```

where

`<accum1, ..., accumn>` is a list of accumulators to be saved, enclosed in angle brackets (<>) and separated by commas. These registers will be saved on procedure entry and restored on procedure exit.

double is an optional parameter that specifies the saving of two-word accumulators. If set to 1, specifies that the FPP is in double mode. The default is zero. The setting does not affect the setting of the FPP; it simply allows the correct computation of the space required for the registers.

Examples:

```
rsave    <ac0,ac4>
```

Save accumulators **AC0** and **AC4** and assume that the FPP is in single mode.

```
rsave    <ac0>,double=1
  or:
rsave    ac0,1
```

Save accumulator **AC0** and assume that the FPP is in double mode.

The 'Begin' Macro

The **begin** macro marks the start of the procedure body. This and the **endpr** macro are the only ones to actually generate code. When the **begin** macro is assembled, all of the data saved up by the previous macros is used to generate procedure entry code and define all of the parameter and variable addresses.

The calling sequence is:

```
begin
```

The 'Endpr' Macro

The endpr macro marks the end of the procedure body. Only one endpr is allowed in each procedure. When the endpr is assembled, registers are restored, the variables and arguments are popped off the stack, and control is returned to the calling procedure. The endpr macro is designed to generate good code for popping the stack and returning.

The calling sequence is:

```
endpr
```

Type Definitions

In addition to the procedure definition macros described above, the PASMAC package defines some standard "types" and provides a set of three macros to simplify the definition of data structures. Each type is represented by its length in bytes.

The predefined types are:

Type	Length
`char`	1
`boolean`	1
`scalar`	1
`integer`	2
`pointer`	2
`address`	2
`real`	4
`double`	8
`procpar`	4

The type `procpar` is actually a record definition having two fields. This type is explained below.

The structure definition package consists of three macros:

Name	Argument	Function
`record`	*typename*	Begins the definition of a record type *typename*. The symbol *typename* will be set to the length of the record at the end of the definition. If the data type is `procpar`, the record is a procedure being passed as a parameter to a MACRO-11 routine.
`field`	*name* *size*	Defines a field in the record. The fields are allocated in ascending order, and any field with a length greater than 1 is allocated on a word boundary. Fields so defined are set equal to the offset of the field relative to the beginning of the structure.
`endrec`		Ends the definition of a record and assigns the total length to the *typename* given in the `record` macro.

For example, consider the following Pascal record definition:

```
prec = record
  Intf1: integer;
  Intf2: integer;
  Boolf1: boolean;
  Realf1: real;
  end;
```

The equivalent code using the structure-definition macros is:

```
record  prec                ; prec = record
field   intf1,integer       ;   intf1: integer;
field   intf2,integer       ;   intf2: integer;
field   boolf1,boolean      ;   boolf1: boolean;
field   realf1,real         ;   realf1: real;
endrec                      ; end;
```

Later in the procedure, where the definition above occurs, we find:

```
var     local,prec          ; local: prec
```

And we would refer to field intf2, for example, as

```
mov     local+intf2(sp),r0
```

The type procpar is used in rare cases when you are passing a procedure as a parameter to a MACRO-11 routine. The definition is:

```
record  procpar
field   pp.proc,address     ; address of the procedure
field   pp.stat,address     ; address of the enclosing static link
endrec
```

This example shows the coding of a MACRO-11 procedure for use with Pascal–2. The procedure chosen for the example is not one that would normally be coded in MACRO-11, but most such procedures are extremely dependent on the operating system. In fact, we begin with a version of the algorithm as it is coded in Pascal-2:

Example: PASMAC - Macro-11

```
{$nomain}

procedure CountOnes(N: integer;           { number to count bits in }
                    var Ones: integer;    { number of "one" bits }
                    var First: integer    { highest "one" bit });
  external;

{ This is a procedure that counts the "one" bits in an integer and
  returns the number of ones in "Ones" and the highest bit found
  in "First".  If no bits are set, "First" receives "-1".
  The procedure uses an extension of Pascal-2 that allows the
  signed number "N" to be treated as an unsigned number "TN".
}
procedure CountOnes;

  var
    TN: 0..65535;                   { local unsigned value of N }
    Bits: 0..16;                    { bit count }

  begin
    First := -1;
    Ones := 0;
    Bits := 0;
    TN := N;
    while TN <> 0 do begin
      if odd(TN) then begin
        Ones := Ones + 1;
        First := Bits;
        end;
      Bits := Bits + 1;
      TN := TN div 2;
      end;
  end;
```

This simple procedure counts the number of bits set in an integer, checking whether the lowest bit is set, incrementing a counter, and terminating when there are no more bits set. The use of unsigned integers (TN, in the range 0..65535), avoids the shifting of the sign bit into the lower-order bits. (Unsigned integers are discussed in the Programmer Reference and in the Language Specification.) This procedure (and many others ~~that~~ are often coded in low-level code) can be coded as a Pascal-2 procedure. But in many ways this procedure is typical of the sort of procedure you may code in MACRO-11:

- It performs a single function with simple internal logic.
- It is a generally useful form of the function, rather than a special use.

- It makes no reference to global variables. All data is passed as parameters.

The first example gives the most direct translation into MACRO-11, with all references to variables made directly to memory. It is quite possible to do the entire function in registers, with some saving in code and execution time, but for the sake of the example we will not do this. We change the algorithm slightly to make use of the state of the condition code at the end of the loop. The use of a conditional branch at this point shortens execution time slightly at no cost in code size.

```
.title  count

; This is a sample procedure that counts the number of bits
; set to one in a word "n" and also sets the variable "first"
; to the bit number of the highest bit set.
;
; This is used strictly as an example; some values that would
; normally be kept in registers are being kept in local variables
; or handled directly in memory for demonstration purposes.

        proc    countones       ; procedure countones(
        param   n,integer       ;   n: integer;
        param   ones,address    ;   var ones: integer;
        param   first,address   ;   var first: integer);

                                ; var
        var     bits,integer    ;  bits: integer; bit counter

        begin                   ; begin
        mov     #-1,@first(sp)  ; first := -1;
        clr     @ones(sp)       ; ones := 0;
        clr     bits(sp)        ; bits := 0;

        tst     n(sp)           ; if n <> 0 then
        beq     10$             ;
1$:                             ;  repeat
        bit     #1,n(sp)        ;
        beq     2$              ;   if odd(n) then begin
        inc     @ones(sp)       ;    ones := ones+1;
        mov     bits(sp),@first(sp)
                                ;    first := bits;
2$:                             ;    end;
        inc     bits(sp)        ;   bits := bits + 1;
        clc                     ;
        ror     n(sp)           ;   n := n div 2;
        bne     1$              ;  until n = 0;
10$:                            ;
        endpr
        .end
```

This procedure illustrates the use of parameters, local variables, and the **begin** and **endpr** macros. The local variable to hold **n** is not needed as there is no distinction made between signed and unsigned integers at the MACRO-11 level. The equivalent Pascal-2 code in the comments should make the MACRO code easy to follow.

In actual practice, local variables would be kept in registers, and the save and rsave macros would be used to save and restore the registers used. The following version is an example of this kind of code.

```
.title  count

; This simple procedure counts the number of bits
; set to one in a word "n" and also sets the variable "first"
; to the bit number of the highest bit set.
;
; Functionally, this is the same procedure as "examp", except that
; it places local variables in registers whenever possible.
;
        proc    countones       ; procedure countones(
        param   n,integer       ;   n: integer;
        param   ones,address    ;   var ones: integer;
        param   first,address   ;   var first: integer);

        save    <r0,r1,r2,r3>

        begin
        mov     n(sp),r0        ; r0 := n;
        mov     #-1,r1          ; r1 := -1; first
        clr     r2              ; r2 := 0; ones
        clr     r3              ; r3 := 0; bits

        tst     r0              ; if r0 <> 0 then
        beq     10$             ;
1$:                             ;  repeat
        bit     #1,r0
        beq     2$              ;   if odd(r0) then begin
        inc     r2              ;    r2 := r2+1;
        mov     r3,r1           ;    r1 := r3;
2$:                             ;   end;
        inc     r3              ;   r3 := r3 + 1;
        clc                     ;
        ror     r0              ;   r0 := r0 shift 1;
        bne     1$              ;  until r0 = 0;
10$:                            ;
        mov     r1,@first(sp)   ; first := r1;
        mov     r2,@ones(sp)    ; ones := r2;
        endpr
        .end
```

In the Pascal program that invokes CountOnes, the following reference is made:

```
procedure CountOnes(N: integer;
                    var Ones: integer;
                    var First: integer);
  external;
```

Placing PASMAC into the System Macro Library

If you often write Pascal programs that invoke MACRO-11 subroutines written using the PASMAC macro package, you might find it desirable to add the PASMAC package to your system macro library. This allows MACRO-11 programs to use PASMAC via the `.MCALL` assembler directive rather than by specifying PASMAC.MAC as a separate input file in the command line.

When you assemble a MACRO-11 subroutine, the assembler searches the system macro library (`LB:[1,1]RSXMAC.SML`) to define macros requested with the `.MCALL` directive. The system macro library normally contains definitions of macros that call system services. You can easily add your own macro definitions to this library.

To add PASMAC to the system macro library, perform these steps:

1. Make a backup copy of your system macro library in case something goes wrong.

2. Using a text editor, create a file called P.MAC, which encloses PASMAC.MAC with a macro definition, as shown below. The definition creates a macro called PASMAC, which contains the entire contents of the file PASMAC.MAC. The macro definition redefines the macro PASMAC to be a null macro. This saves space and time in the assembler. The macro definition must be in upper case.

```
.MACRO   PASMAC
:
:
:————————— contents of PASMAC.MAC
.MACRO   PASMAC
.ENDM

.ENDM   PASMAC
```

   We offer two ways to create P.MAC. The first way is to create the skeleton macro, then insert the contents of PASMAC.MAC at the location shown above. The other possibility is to first copy PASMAC.MAC to P.MAC and then edit P.MAC, placing the skeleton macro directives around the contents of PASMAC.MAC.

3. Add the contents of the file P.MAC to your system macro library. The command to use is:

```
>LBR LB:[1,1]RSXMAC.SML/IN=P.MAC
```

PASMAC.MAC is now a part of the system macro library. You can now use the `.MCALL` directive to define the PASMAC macros in your

MACRO-11 routines. This is illustrated below. The routine simply clears registers R0 and R1.

```
        .title  test
        .mcall  pasmac                  ; Read the PASMAC macro
        pasmac                          ; Define PASMAC macro

        proc    test                    ; Sample procedure
        param   foo,integer
        save    <r0,r1>
        begin
        clr     r0
        clr     r1
        endpr
        .end
```

Now you can assemble the routine without specifying the PASMAC.MAC source file on the MAC command line.

```
>MAC TEST=TEST
```

PROSE: A Text Formatter

Computerized text-processing tools such as text editors and formatters can ease the tedious preparation and editing of computer-oriented documentation. Instead of cutting, pasting, and retyping hard copy, you instruct the computer to insert changes, reformat and number pages, then reprint the document. PROSE, a text-formatting utility program, allows you to print any document in a variety of formats.

This guide describes the operation of PROSE, providing an overview of text-formatting procedures for PROSE and a detailed explanation of PROSE directives. The first-time user of PROSE may read this guide from beginning to end, using it as a tutorial while producing a first document. The more experienced reader may use it as a reference source. As an aid to both, this guide groups PROSE directives by function into four sections: controlling input, establishing format, indexing, and printing. The order of the directives in the guide reflects the order in which these directives may be applied to a text. The reader should have some basic knowledge of a text editor.

PROSE requires a small number of easily learned commands. Unlike some text-formatting programs, which use macro commands, variables, and other features usually associated with programming languages, PROSE does not overwhelm the user with complicated syntax. The text stands out, not the directives. This simplicity allows you to produce high-quality text with a minimum of effort.

Like many text formatters, PROSE formats text in pages, filling and justifying lines, placing titles and page numbers as needed. The next table shows some common features of text formatters that PROSE does and does not have.

Prose Can ...	... And Cannot
Underline	Control photo-typesetting machines
Hyphenate words	Do graphics
Convert upper-case input to mixed-case output	Produce multi-column text
Produce a sorted index	Store text and retrieve it later
Print selected pages	Use tabs

PROSE may or may not be the tool for a given application.

PROSE Basics

The basic units of any text-formatting system are the word, the line, and the paragraph. In PROSE, a `word` is defined as any non-blank string of characters, with a blank on either side. For the purposes of formatting, a punctuation character is part of the word next to it. A `line` consists of the number of words that PROSE fills between margins. PROSE places as many words as possible into each output line, adding blanks to justify the lines to left and right margins.

Text formatting is largely `filling` and `justifying`, a process illustrated by the next example.*

Input to PROSE:

```
  Eddie went to the ground floor cafeteria and got a sandwich
and container of coffee, then went back to his office to work on
the water bill survey. No one else was there; the others were still
out on their regular lunch hour.
  Why not? he asked himself. It took only ten minutes from start to
finish: eight to find the code, one to decide how to do it, and one
more to type the orders into the computer console. When he finished,
the screen told him that the violation number had been removed.
  A few minutes later his office door opened. It was his boss. He was
back ten minutes early from his lunch hour.
  "You're here," the boss said.
  "That's right," Eddie said.
  "Good," the boss said, leaving without bothering to close the
door.
```

Output from PROSE:

```
  Eddie went to the ground floor cafeteria  and  got  a  sandwich  and
container of coffee, then went back to his office to work on the water
bill survey.  No one else was there; the  others  were  still  out  on
their regular lunch hour.
  Why  not?  he asked himself.  It took only ten minutes from start to
finish: eight to find the code, one to decide how to do  it,  and  one
more  to type the orders into the computer console.  When he finished,
the screen told him that the violation number had been removed.
  A few minutes later his office door opened.  It was  his  boss.   He
was back ten minutes early from his lunch hour.
  "You're here," the boss said.
  "That's right," Eddie said.
  "Good," the boss said, leaving without bothering to close the door.
```

When the user gives no special instructions, called `directives`, PROSE operates in the default mode as shown in the example above. In the default mode, PROSE automatically fills and justifies output lines, formatting the output into pages. Directives instruct PROSE to do anything more sophisticated. If the user doesn't enter certain directives, PROSE supplies their default forms.

---

* Text examples in this guide are excerpted from *The Programmer* by Bruce Jackson. Copyright © 1979 by Bruce Jackson. Reprinted by permission of Doubleday & Company, Inc.

Structure of Directive Lines

In general, a directive line has three components: the escape character, the directive name, and the parameter for its application. Most PROSE directive lines take one of three forms:

```
.directive name
.directive name integer
.directive name( parameter )
```

The **directive escape character** is placed in the first column of an input line to indicate that at least one directive follows. The period (.) is the default escape character because it seems unlikely that anyone would want to type a period in the first column of a line of text. The default may be changed with the INPUT directive (see "Controlling Input to PROSE").

After the escape character comes the name of the directive that PROSE is to execute. The *directive name* may be abbreviated to three letters (in fact, PROSE only examines the first three). Examples in this guide show directives typed in upper case, but PROSE accepts both cases.

The directive name may or may not be followed by a *parameter*. The BREAK directive, for example, doesn't require a parameter. If a necessary parameter is omitted, PROSE supplies a default value for that parameter. The default values that PROSE uses are listed in a table of options under each directive.

A parameter may be one of three types:

- Text on the remainder of the directive line.
- An integer.
- Any specific options enclosed in parentheses, consisting of other directive names, integers, or keywords defined by the directive itself.

In text-processing systems such as PROSE, a **keyword** (also called a descriptor) categorizes or indexes information. Many PROSE directives use special letters or characters to express the options assigned, as do the INPUT or FORM directives. In directives such as RESET(MARGIN), the keyword is the name of the directive to be changed. The summary directive table in Appendix A indicates the parameter type that each directive may take.

Numeric values used as a parameter or part of a keyword may be either an explicit positive integer or a relative value. A relative value, specified by a plus or minus sign before the integer, indicates that the old value should be increased or decreased by the amount of the integer. For example, if the left margin is set to 10 and the right margin to 70, you may use a relative values in the directive

```
.MARGIN( L+5 R-5 )
```

to squeeze the margins together by 5 characters on each side.

Placement of Directives

Directive lines are usually separated from lines of text (see the next sample file). Several directives may be typed on the same line, provided that they are separated by the directive escape character, as follows.

```
.BREAK.SKIP 2.MARGIN( L5 R65 )
```

Some directives take the remainder of the line as their parameter, so no other directives may follow these (e.g., COMMENT directive). The following sample shows the placement of PROSE commands in an input file.

Input to PROSE:

```
.COMMENT  This example makes very primitive use of directives,
.COMMENT  but it produces exemplary text.
.MARGIN( L10 R60)
.INDENT 2
Eddie ordered that the tax roll and Yellow Pages tapes be returned
to storage. A few seconds later the video screen told him they had
been returned to their appropriate storage locations.
.BREAK.INDENT 2
Eddie smiled at the screen. He loved the computers. They would do
exactly what you told them to do and they would never lie to you.
Two inhuman characteristics. People who complained about the inhumanity
of computers were right. They didn't know how to care or betray.
.BREAK.INDENT 2
The next operation was more complicated. He had prepared for
it some time before, and the preparation had required many
separate inquiries.
```

Output from PROSE:

```
  Eddie ordered that the tax roll and Yellow Pages
tapes be returned to storage.  A few seconds later
the video screen told him they had  been  returned
to their appropriate storage locations.
  Eddie  smiled  at  the  screen.   He  loved  the
computers.  They would do exactly  what  you  told
them  to  do and they would never lie to you.  Two
inhuman characteristics.   People  who  complained
about  the  inhumanity  of  computers  were right.
They didn't know how to care or betray.
  The next operation was more complicated.  He had
prepared for it some time before, and the prepara-
tion had required many separate inquiries.
```

For more sophisticated examples, see "Appendix B: Examples of PROSE Directives in Text."

A long directive may extend beyond one line. A continued line is indicated by a **continuation character**, a plus sign '+' placed in column one. The next example shows suggested placement of the continuation character:

```
.FORM( [ /// L58 // #73 'PAGE' p /// ]
+      [ /// L58 //     'PAGE' p /// ] )
```

Generally, directives are placed at the beginning of the input file or at the point in the text where the directive takes effect. Most directives either control the functions of PROSE or set general format guidelines for the document. These are placed at the beginning of the text and their operations are applied throughout, unless temporary changes are made. The **FORM** directive, for example, establishes page format for the rest of the text, but the number of lines per page may be adjusted by an option of the **PARAGRAPH** directive. Directives that apply only to a particular line, such as **INDENT**, **BREAK**, and **COMMENT** in the sample above, are placed wherever necessary throughout the text.

Running the PROSE Program

No actual formatting takes place until the input file, containing text and directive lines, is submitted to PROSE for processing. You create the input file with your system's text editor. PROSE places the formatted text in an output file for submission to a printer or for display on a terminal screen.

To format the input file, invoke PROSE as shown:

C:> `prose` *input-file[/*`output=`*output-file]*

where

input-file is the PROSE source file(s). Multiple input files are read and concatenated from left to right.

`/output=`*output-file* is the formatted PROSE file. The `/output` switch is optional. If it is omitted, an output file, with the same name as the rightmost input file and the extension .DOC, is placed in the current directory.

The output file may then be printed.

Header Files

Certain directives nearly always appear at the head of any input file. If many documents use these directives in the same form, you may set up a header file rather than the directives into each document's input file. Header files also provide an easy way to choose among various forms or output devices.

As a general practice, we recommend that you set up each PROSE text without OUTPUT or FORM directives. Instead, keep these directives in another file that you use as the first input file, or "header file." For example, the next header file, `printer.prs`, contains a set of OUTPUT and FORM directives for the line printer.

```
.output (lpt s10 e+ u-)
.form( [  t #62 e /// 154 /// #33 '- ' pa:1 ' -' /// ] )
```

If this header file is stored in the directory `\usr\include`, you use this command to prepare the document for the line printer:

```
C:> prose \usr\include\printer.prs  usrman.prs/output=lastone.doc
```

where

`printer.prs:` is the header file containing general formatting directives.

`usrman.prs:` is the input file.

`lastone.doc:` is the output file.

PROSE prints the output file according to directives in the header file. See "Page Format" and "Specifying Output Devices" for the functions of the individual directives included in the header file.

Controlling Input to PROSE

The directives in this section control the input to the PROSE program. Generally, they are placed at the beginning of the input file for a document or in a header file to be used for all documents. You may set and change them as needed throughout the text.

INPUT Directive

The INPUT directive tells PROSE how to interpret certain control characters in the input file and sets the maximum length for input lines.

The next table summarizes the options for its parameter.

Key Letter	Meaning	Type	Default
B	Explicit blank character	character	nul
H	Hyphenation character	character	nul
C	Case-shift character	character	nul
U	Underline character	character	nul
D	Directive escape character	character	.
W	Input width	number	150
K	Keep	number	next

The options, which may be given in any order, consist of a key letter followed by a value. Unless the user specifies both the key letter and a value, the default value is assigned when PROSE begins processing. A value in the parameter changes only when a new value is given. No INPUT option uses relative values.

B: The explicit blank character indicates a blank that PROSE should treat as if it were a character. With the cross-hatch '#' specified as the explicit blank, the next example shows how two words separated by an explicit blank are never split from one line to the next. PROSE never fills blanks between the words to justify a line.

```
.INPUT( B# )
...someone like the imaginary Dr.#Conrad and...
```

H: The hyphenation character defines hyphenation points within words. Sometimes a long word causes many blanks to be inserted to justify the preceding line. PROSE hyphenates such a word if you have defined the syllable boundaries within it. Of course, not all the syllable boundaries need be specified, only those at which you want PROSE to be able to split a word. For example, if the hyphenation character is the slash '/', you may type "syncopation" as syn/co/pa/tion. PROSE inserts a hyphen '-' only when the characters on both sides of the hyphenation point are letters. This restriction allows you to type "hyper-active" as hyper-/active, and PROSE splits the word if necessary, without adding a superfluous hyphen. If PROSE is forced to insert blanks beyond a certain threshold set by the OPTION directive, PROSE issues an error message on the line that needs hyphenation characters.

C: To produce mixed-case output from upper-case-only input, you must specify a case-shift character in the INPUT directive parameter, causing PROSE to automatically shift all upper-case letters to lower case. To preserve certain upper-case letters, such as initial capitals for names and sentences, surround the letter or letters with case-shift characters. PROSE shifts to upper case for all characters between case-shift characters. "Stuttering" is another way to designate capitals among upper-case-only input. Since most upper-case letters are at the beginning of a word (following a blank), you use two letters to indicate a single capital. Words that already begin with a double letter produce a single capitalized letter unless you put two case-shift characters before the word.

Input to Pr:	Output From Pr:
^^LLAMA	llama
^^OOPS	oops
LLLAMA	Llama
OOOPS	Oops

The next example demonstrates both ways of producing mixed-case output from upper-case-only input. The case-shift character is easier to use for long strings, such as example programs, that are to be capitalized. Stuttering is easier when you want to capitalize a single character, such as the first word of a sentence. You may use both methods in the same text as shown below.

Input to PROSE:

```
.INPUT(C^)
HHE HAD EIGHTEEN MINUTES, PLUS THE LOCAL CONNECTION. TTWENTY-ONE
MINUTES. AA WORLD OF TIME ON A COMPUTER. HHE WAS READY WITH HIS
QUESTIONS.
  ^WHAT IS CODE FOR PROGRAM?^
  ^COMPUTER BANDIT.^
  ^WHAT IS KNOWN ABOUT COMPUTER BANDIT?^
  TTHE SCREEN RAPIDLY FILLED WITH THE DATES AND AMOUNTS OF HIS
AAMERICAN EEXPRESS THEFTS, SOME OF HIS RECENT INFORMATION SCANS,
THE REPORTS TO THE NEWSPAPERS IN NNEW YYORK.
```

Output from PROSE:

```
He had eighteen minutes, plus the local connection.  Twenty-one
minutes.  A world of time on a computer.  He was ready with his
questions.
  WHAT IS CODE FOR PROGRAM?
  COMPUTER BANDIT.
  WHAT IS KNOWN ABOUT COMPUTER BANDIT?
  The screen rapidly filled with the dates and amounts of his American
Express  thefts,  some of his recent information scans, the reports to
the newspapers in New York.
```

For conversion of mixed-case input to upper-case-only output, see the OPTION directive.

U: Text surrounded by the underline character are underlined. Blanks are not underlined, but explicit blanks are.

D: The directive escape character is placed in the first column of an input line to flag it as a directive. Use this option only to define a directive escape character other than the period.

W: The input width W specifies the number of characters to be read from each input line. Users need to change only the input width for special jobs.

K: The keep option explicitly specifies the keep buffer to be used to store the new input options. By default, PROSE uses the numerically next buffer. (See "Changing The Format Control Directives" for detailed explanation of the use of keep buffers in PROSE directives.)

OPTION Directive

The OPTION directive gathers together miscellaneous options that affect the filling and justifying PROSE does during text formatting. These options are summarized in the next table. Key letters are followed by a switch symbol (+/-) or an integer, as shown in the default column. For the switch-type options, the plus sign '+' means on and the minus sign '-' means off.

Key Letter	Meaning	Default
E	Print error messages	+
J	Justification limit	3
F	Fill output lines	+
L	Left justify	+
R	Right justify	+
S	Spacing	1
M	Multiple blanks	+
P	Two blanks after periods	+
U	Shift to upper case	-
K	Keep	next

As processing begins, PROSE assigns the default value for each option without a specified value. A parameter value changes only when a specification is given. No option uses relative values.

E: Error messages appear in the formatted text of the main output files at the approximate location of the errors. Error messages are suppressed when this option is off (`E-`).

J: PROSE inserts blanks as needed to justify the left and right margins of an output line. The `justification limit` controls the point at which PROSE attempts to hyphenate a word. If, for instance, the justification limit is set at 3, then the hyphenation process is invoked when PROSE has to insert three blanks between adjacent words on a line. If hyphenation is not possible, or PROSE is not able to bring the number of inserted blanks below the limit, an error message is printed for the line(s).

NOTE

Settings for options `E` and `J` may be varied according to the draft you are working on. Setting `J` to an arbitrarily high number (e.g., 20) and turning off `E` helps avoid hyphenation errors until you are ready to deal with them, usually in the later stages of document preparation.

F: Output lines are automatically filled and justified as described in the "PROSE Basics" section. If the `fill` option is off, PROSE prints the input lines as they are, without reformatting to fill the output lines. In effect, a justification break is done after each input line. Option `F-` is most useful for literal text, such as program examples, where spacing between words must be exactly as typed.

L:
R: The `left` and `right justify` switches work together to determine the justification to be done. If both options are on, output lines are justified to both the left and right margins. If both options are off, the lines are centered between the two margins. If one is on and the other is off, one margin (either left or right) is straight and the other ragged. The next examples demonstrate the output from the four combinations.

Output from `.OPTION( L+ R+ )`:

```
Eddie did four more operations, three of  them  involving  county  and
city payroll checks, all of which were handled by the same computer.
```

Output from `.OPTION( L- R- )`:

```
 He gave an across-the-board raise of fifty dollars per check to all
teachers in the ghetto schools.  He deducted an equivalent amount from
      the checks of the city's highly paid political appointees.
```

Output from `.OPTION( L+ R- )`:

```
Then he erased the tapes for outstanding private residential water
bills.  People, he decided, shouldn't have to pay for a drink of water
or to be able to flush ... a toilet.
```

Output from `.OPTION( L- R+ )`:

```
 The final operation was one he hadn't thought of—earlier; it had come
          to him during the night's work.  It was easy enough with the
                                            information he now had.
```

S: The spacing option 2 generates double-spaced output; the spacing option 3 generates triple-spaced output. By default, text is single-spaced.

M: If the `multiple blanks` option is on (`M+`), multiple blanks in the input file are considered to be significant. That is, if several blanks are placed between two words in the input file, at least that many appear in the output file; PROSE may add blanks during justification. If the option is off, multiple blanks are treated as a single blank.

P: The `2 blanks after periods` option places at least two blanks after every period. PROSE does not add blanks before justifying if two are already present. This makes for consistent spacing in the final copy even if you are not careful about typing 2 spaces after sentence periods in the original. Three or more blanks after a period are treated as multiple blanks.

U: For output devices that cannot process mixed-case files, the `shift to upper case` option shifts all lower-case letters to upper-case letters. This option is also useful for printing an entire passage or example, such as a sample program, all in upper case. For conversion of upper-case-only input to mixed-case output, see the `INPUT` directive.

K: The `keep` option explicitly specifies the keep buffer to be used to store the new options. By default, PROSE uses the numerically next buffer. (See "Changing Format Within the Text" for use.)

Setting Up the Document's Format

This section explains the use of directives to format text. These directives specify page format, margins, paragraphing conventions, justification breaks, and blank or comment lines.

Page Format

The FORM directive defines the page format, including insertion of titles, date/time, blank lines, page numbers, and other textual items at the top or bottom of the page. The FORM directive works with the COUNT, TITLE, and SUBTITLE directives. The PAGE and PARAGRAPH directives override the page-break function of the FORM directive.

FORM Directive

The FORM directive produces ~~a variety~~ of page formats, depending upon the options specified in its parameter. The table below contains the available FORM options; the following paragaphs explain their use.

Key Char	Meaning	Default Field Width
[	Define top of page	–none–
]	Define bottom of page	–none–
#*n*	Tab forward or backward to absolute column *n*	–none–
S	Subtitle	its length
T	Main title	its length
L*n*	Fill in *n* lines of running text on the page	–none–
/	Print an end of line (by itself, a blank line)	–none–
/*n*	Print *n* ends of lines	–none–
P*f*	Current page number, *f* selects the form:	3
	N or n Arabic numerals (default)	[The field width
	L Upper-case letter	is expanded if
	l Lower-case letter	needed]
	R Upper-case Roman numerals	
	r Lower-case Roman numerals	
'...'	Print material within quotation marks as literal text	–none–
C	24-hour clock as hh.mm.ss (e.g. 15.37.58)	8
D	Raw date as yy/mm/dd (e.g. 82/02/13)	8
E	Nice date as dd Mmm yy (e.g. 13 Feb 82)	9
W	Wall clock as hh:mm PM or hh:mm AM (e.g. 3:37 AM)	8

If the FORM directive is omitted completely, PROSE uses the default form:

```
.FORM( [ // T #62 E /// L54 /// #33 '- ' PN:1 ' -' //// ] )
```

The sequence of options within parentheses corresponds to the format of the page from top to bottom. The FORM directive builds text lines from left to right, starting in the first printable column unless a tabbing specification (#n) starts text at a specific column.

FORM directive parameters generally begin and end with the definition characters for a top-of-page and bottom-of-page. The top-of-page definition '[' has several uses. You may direct PROSE to send a page eject to the output device when it reaches the top of a page. Also, you may request a pause at the top of each page to allow you to change paper on the printer (see information on the OUTPUT directive in "Printing the Document"). At the end of the document, PROSE signals one last page eject and continues to interpret the FORM specification until it reaches another top-of-page. This ensures the execution of any commands specified for the bottom of the last page, such as a page number. PROSE increments the page number at the bottom-of-page character ']'. So, if you print the page number both before and after the bottom-of-page definition, you get different numbers.

To print slightly differing formats for facing pages, specify a format for each page between a pair of page definition characters. For example, this form directive prints the page number at the bottom right of odd numbered pages and at the bottom left of even pages.

```
.FORM ( [ // T #62 E /// L56 // #63 'PAGE' P /// ]
+       [ // T #62 E /// L56 //      'PAGE' P /// ] )
```

Appendix B contains another example of the FORM directive used to print facing pages.

Page length is determined by the specification for the number of lines. PROSE breaks pages at the number of lines set by the FORM directive's Ln specification, unless the PAGE directive or the optional automatic page eject for the PARAGRAPH directive is used (see "Page Breaks"). If the Ln specification is omitted entirely, PROSE supplies the default value of 54 lines per page. If the FORM directive parameter contains a key letter L without a value, no special page formatting is done. Page length is infinite, which is useful for working with documents on terminals, where pages are irrelevant. In this mode, a PAGE directive with no parameter puts 5 blank lines between sections of text.

Titles, subtitles, page numbers, and dates are placed in fields at the top or bottom of the page. Although default values are sufficent for most situations, the field width may be set to a particular value by placing a colon and the value after the key letter. For example, T:30 prints the title in a field of 30 characters. Specified field widths are sometimes useful for truncating long titles. PROSE fills the field from right to left. The tabbing specification #n places the field horizontally on the page.

The FORM argument is re-scanned as each page of output is produced, so that any change in a title buffer made with the TITLE or SUBTITLE directive causes the new title or subtitle to be inserted on the next page. The TITLE directive enters the remainder of the line into the main title buffer. The FORM directive uses the contents of the title buffer to print a title on the page as specified. The SUBTITLE directive enters the remainder of the line into the subtitle buffer, to be used by the FORM directive to print a subtitle on the page as specified. See examples in Appendix B.

PROSE adds a blank line for each '/' mark in the FORM directive. These blanks are placed uniformly on each page, in the relative position that they appear in the directive, usually at the top and bottom, between the body of the text and the title and page number. The alternative /n allows a shorter expression of numerous blank lines; either form may be used.

Page numbers are incremented by the page counter and placed on the page by the Pf option of the FORM directive. The COUNT directive sets the page counter. The integer in the COUNT parameter may be a relative value; for example, .COUNT +1 increments the page number by one. By default, the page counter sets the page number to 1.

The literal text option allows you to add such touches as hyphens surrounding the page number (see default FORM directive) or other text that must appear exactly as typed. For example, suppose a press release required the word "more" with parentheses around it at the bottom of each page. You may use the literal text specification as follows:

```
.FORM( [  T /// L54 /// #28 '(more)' /// ] )
```

PROSE users may choose among four styles of dates. The date is placed on the page in much the same manner as a title.

Page Breaks

The PAGE directive signals a page eject when fewer than the specified number of lines remain on the current page. If no parameter is given, the PAGE directive does an unconditional page eject. The PARAGRAPH directive's automatic page eject includes the page-break function in the paragraph format for the document. .PAGE 3 and .PAR(P3) are equivalent, except that .PAGE 3 must be explicitly placed by the user, while PROSE executes .PARAGRAPH (P3) wherever indicated by the paragraph flag character.

Margins

The MARGIN directive sets the left and right margins for filling and justifying. The value for left margin indicates the column in which the line of text begins; the right margin value is the column number of the last printed character. Thus, subtracting the left margin from the right margin gives the number of columns for printed text.

The options, which may be given in any order, consist of a key letter followed by a value. The next table lists the key letter for each option.

Key Letter	Meaning	Type	Default	Relative
L	Left margin	integer	0	yes
R	Right margin	integer	70	yes
K	Keep	integer	next	no

Margins are set before text processing begins. PROSE assigns default values of L0 R70 if no MARGIN directive is used or if the directive is given without a parameter. A value changes only when a new specification is given. The **keep** option explicitly specifies the keep buffer to be used to store the new margins. By default, PROSE uses the numerically next buffer.

The **INDENT** directive moves the next line of text to the right of the page by the given number of spaces. When the directive is used without a parameter, the default value is 5. The **UNDENT** directive moves the next line a certain number of spaces to the left. (The undent is sometimes known by the name "outdent" or "hanging indent.") If the given parameter would undent the text past the leftmost column of the printed page, the directive undents only to the leftmost printable column. If no parameter is given, the default undents to the leftmost printable column.

Paragraphs

Although you may use any justification break methods to distinguish between one paragraph and the next, the **PARAGRAPH** directive provides a more versatile method of creating paragraphs.

Placed at the beginning of the text, the directive sets the general form of paragraphs and specifies a **paragraph flag character**. Many **PARAGRAPH** options take the place of other directives, so the directive is a powerful tool.

When the paragraph flag character is placed in the first column of a text line to signal a new paragraph, PROSE takes any of the following actions that are specified by the parameter.

Key Letter	Meaning	Type	Default
F	Paragraph character	character	nul
I	Automatic indent	number	0
U	Automatic undent	number	0
N	Number generator		–none–
P	Automatic page eject	number	0
S	Automatic skip	number	0
K	Keep	number	next

When it begins processing, PROSE assigns the default value for each option in the **PARAGRAPH** directive parameter. If the input file contains no **PARAGRAPH** directive, or if an option is not specified, the default value is used. No **PARAGRAPH** option uses relative values.

By manipulating the options in the parameter, you may direct PROSE to take any of the following actions for paragraphs.

F: The **paragraph flag character** invokes a collection of paragraphing actions when it appears in the first column of an input line. Note that this character must be set in the first **PARAGRAPH** directive, or no other options apply. As the only specified option, the paragraph flag character signals a justification break.

I: U: The **automatic indent** or **automatic undent** applies to the first line of the paragraph and moves the line left or right a given number of spaces. If the number generator is used, the indent or undent is applied after the number is generated (see the example using both options below).

N: The **number generator** produces a new number (or letter) for each occurrence of the paragraph flag character. PROSE inserts the number in lieu of the paragraph flag character when the line is formatted, so you must put a space between the paragraph flag character and the text line, if you want one to appear in the output. The number generator is initialized to 1 each time new paragraph settings go into effect. Resumption of an old setting also resumes the old numbering. The number generator's keyword contains these fields (spaces not allowed):

N *numeric-field field-width*

The key characters for *numeric-field* are:

-blank-	No numbering
N or n	Arabic numerals
L	Upper-case letter
l	Lower-case letter
R	Upper-case Roman
r	Lower-case Roman

The field width for the numeric field, expressed as an integer, is expanded if necessary. If, for example, you want an Arabic numeral with three spaces left for the numeral, the keyword is Nn3.

The folowing input and output examples illustrate one style of numbered, undented paragraph created with the automatic undent and number generator options. Note that the margin adjustment places the paragraph number in the leftmost column of the printed page and that the ampersand (&) has been specified as the paragraph flag character.

Input to PROSE:

```
.MARGIN( L10 )
.PARAGRAPH( F& Nn1 U5 )
& Eddie worried about that one for a while, then came up with
a very simple answer:  he would ask the computers if they had
any such self-inspection instructions in their programs.  It
would become part of his regular greeting:  HELLO. HOW ARE YOU
AND ARE YOU PROGRAMMED TO TRAP ME?
```

Output from PROSE:

```
1 Eddie worried about that one for a while, then came up  with  a
     very  simple answer:  he would ask the computers if they had
     any such self-inspection instructions in their programs.  It
     would  become part of his regular greeting:  HELLO.  HOW ARE
     YOU AND ARE YOU PROGRAMMED TO TRAP ME?
```

P: The automatic page eject simulates the effect of the PAGE directive. For instance, the directive .PAR(P4) causes PROSE to eject a page if fewer than four lines of the paragraph are left at the bottom of the page. The command is applied after the automatic skip.

S: The automatic skip functions the same as a SKIP directive, placing a blank line before the first line of the paragraph.

K: The keep option explicitly specifies the keep buffer to be used to store the new paragraph options. By default, PROSE uses the numerically next buffer.

Values and options may be changed for particular paragraphs or sections of the document, as explained in "Changing Format Within the Text."

Comments

Using the COMMENT directive, you may include information in the source of a document that is not printed in the formatted copy. As shown in the examples in Appendix B, PROSE treats the remainder of the line as a comment and ignores it.

Changing Format Within the Text

To make the fullest use of PROSE, the user must manipulate such options as blank characters, spacing, margins, or page breaks within the text. The BREAK and SKIP directives allow you to interrupt the established formatting process.

At certain points, you may need to switch formats for specific situations, such as example programs or blocked quotations. OPTION, MARGIN, and PARAGRAPH settings may change frequently, but the number of different settings is probably predictable and few. Depending upon the number, the variety, and the frequency of changes the text requires, the following techniques may help you to enter new directives or restore previously used options.

Breaking and Skipping Lines

One of the simplest and most frequently used instructions, a `justification break` causes PROSE to stop filling the current output line and print it without justifying. A line break may be indicated in many ways. Text may be separated (broken) by one or more blank lines inserted in the text, by leading blanks typed on an input line (a paragraph indention), or by the `BREAK` directive. The next example illustrates these three methods.

Input to PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
.BREAK
"Right," Barstow said.
.BREAK
"It's like Battleship," Purvey said, smiling at Barstow
and the IBM man.

"You understand perfectly," the IBM man said, "That's
exactly what it is."
```

Output from PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
"Right," Barstow said.
"It's like Battleship," Purvey said, smiling at Barstow  and  the  IBM
man.

"You  understand perfectly," the IBM man said, "That's exactly what it
is."
```

With any of these methods, you direct PROSE only to do a justification break. PROSE does not skip lines or indent unless you explicitly enter blank lines or indentions in the input file.

The `SKIP` directive prints blank lines within the text by skipping a certain number of output lines. `SKIP` does not print blank lines at the top of a page, unless you enter at least one actual blank line before the `SKIP` directive. The default value of the `SKIP` directive is 5 lines.

Keep Buffers

The **keep buffer** is a simple way to change directives that control input or format. Each time a change in one of these directives is processed, PROSE saves the new values in a **keep buffer**. Ten keep buffers (0 through 9) are associated with each directive. You may use a keep parameter to specify the buffer to be used; if no buffer is specified, the values are saved in the numerically next buffer. To recall a previously used value, you enter the directive with the number of the keep buffer as the parameter.

For example, suppose that a double-spaced text has a number of paragraph-length quotations, which are to be typed as single-spaced blocks indented ten spaces from each margin. Using the keep option in the parameters for the **INPUT**, **MARGIN**, and **PARAGRAPH** directives, you may store the format specifications for each situation in two different keep buffers by entering these directives at the beginning of the input file:

```
.OPTION(K1 S2).MARGIN(K1 L10 R70).PARAGRAPH( K1 F& I2 S1 )
```

Then you enter these directives for the new format

```
.OPTION(K2 S1).MARGIN(K2 L20 R60).PARAGRAPH( K2 F& I0 S0 )
```

before the first blocked paragraph. To resume the standard paragraph format, you then enter the directive names with the number of the keep buffer. The input and output files look like this:

Input to PROSE:

```
.OPTION( K1 S2 ).MARGIN( K1 L10 R70).PARAGRAPH( K1 F& I2 S1 )
&There was nothing to be done.  He had once heard a comedian
say,
.OPTION( K2 S1 ).MARGIN( K2 L20 R60 ).PARAGRAPH( K2 F& I0 S0 )
If you don't like the telephone company,
you know what you can do? Two tin cans and
a piece of string, that's what you can do.
That's the only alternative you've got.
.OPT 1.MAR 1.PAR 1.SKIP 1
The people in the audience had laughed. Eddie thought about
it now and decided it wasn't funny at all.
```

Output from PROSE:

```
  There was nothing  to  be  done.   He  had  once  heard  a

comedian say,

          If you don't like the telephone company,
          you know what you can do? Two  tin  cans
          and  a  piece of string, that's what you
          can do.  That's  the  only  alternative
          you've got.

The people in the audience had laughed.  Eddie thought about

it now and decided it wasn't funny at all.
```

To change format for the next single-spaced, blocked paragraph, you only need to enter:

```
.OPT 2.MAR 2.PAR 2
```

The example is a little more cumbersome than is necessary for one format change. Actually, you need only enter `.OPT.MAR.PAR` to return to keep buffer 1 in the text shown above. When no parameter is specified, the values are set to those stored in the numerically previous keep buffer, since the keep number is automatically incremented whenever a directive is entered and automatically decremented when that directive is entered without a parameter. Used in this way, the keep buffers function as a "stack" for temporary storage of variations from a basic format.

Reset Directive

The **RESET** directive sets twelve frequently changed directives to their default values. The next table summarizes the effect of the **RESET** directive on each:

Directive	Effect
INPUT	Default values for all options.
OPTION	Default values for all options.
FORM	Default values for all options, causes page eject.
COUNT	Sets page counter to 1.
TITLE	No titles until reentered.
SUBTITLE	No subtitles until reentered.
PAGE	Causes a page eject.
INDEX	Deletes all accumulated entries.
MARGIN	Default values for all options.
PARAGRAPH	No paragraphing until directive reentered.
OUTPUT	No pause at top of page; carriage return to do underlining; causes page eject.
SELECT	Discontinues page selection; all pages to be printed.

The **RESET** directive may be used three ways:

- Entering the directive name with no parameter resets the values of all directives to their defaults:

```
.RESET
```

- Using a directive name as a keyword resets the selected directive. For example, this command resets only the **MARGIN** and **OPTION** directives:

```
.RESET( MARGIN OPTION )
```

- Stating the keyword "except" and a directive name in the parameter excludes a selected directive. For example, this directive resets all directives with the exception of FORM and OUTPUT:

```
.RESET( EXCEPT FORM OUTPUT )
```

New directives may be entered after the RESET directive. The RESET directive is an easy way to clear a complicated series of format changes from the keep buffers.

Creating an Index

The INDEX and SORTINDEX directives provide the information PROSE needs to create an index for the document. The INDEX directive is entered as .INX to distinguish it from .INDENT and takes the remainder of the line together with the current page number as an index entry. As the formatted text migrates from page to page in various drafts, the page numbers in the index are updated.

Index entries accumulated by INX directives may be sorted alphabetically or by page number, then printed in a relatively flexible manner. The SORTINDEX directive allows you to specify the method of sorting entries and the format for printing the index.

The options for the SORTINDEX directive, listed in the next table, may be given in any order.

Key Letter	Meaning	Default
S	Sorting option. If this is numeric, it is the first significant column for alphabetic sorting. If it is the letter P, sorting is selected by page number.	1
M	Margin (left margin before index line)	0
P	Column (in index entry) to insert page number	0
L	Left width of page number (field width of number)	2
R	Right width of page number, blanks printed after	2

In the absence of a parameter, default values are used.

Printing the Document

This section does not anticipate all the possible idiosyncracies of the processor and its peripherals; it gives you directions for printing all or part of the document on certain standard devices.

Specifying Output Devices

The OUTPUT directive defines important aspects of the output device to which you send the formatted text. The directive takes the general form:

```
.OUTPUT(device,options...)
```

One of the following acronyms indicates the output device to be used.

ASC	ASCII terminals use the backspace for underlining, but are otherwise the same as the lineprinter (LPT) below. Pauses for page eject, however, are handled differently (see the P option below).
LPT	Line printers use overprinting with a carriage return to do underlining. This is the default output device.

The next table contains the options for each output device. Keyword type is an integer or a switch.

Key Letter	Meaning	Default
E	Page eject at top of page ([in FORM description)	-
P	Pause at top of page	-
S	Shift output lines to the right	0
U	Underlining is available	+

These options may be given in any order.

E: The page eject option prints a form feed character for each time PROSE reads [in the FORM specification.

P: The pause option causes PROSE to stop printing and await operator acknowledgement each time a '[' character is encountered in the FORM specification. On an ASC terminal, PROSE sounds the bell and waits for a carriage return to be entered. For an LPT output device, no action is taken.

S: The shift option shifts all PROSE output to the right by any number of spaces up to 50. This makes it easy to center output on a wide printer page.

U: If the destination terminal does not have underlining capability and the input file contains underline characters, the underlining available option should be turned off to prevent PROSE from trying to generate overprinted underlines.

Usually, the OUTPUT directive appears only at the beginning of the input file or in a header file. However, it must also be used immediately after a .RESET(OUTPUT) directive.

The LITERAL directive is useful for producing special printer control characters on some systems. It prints the remainder of the input line as a single output line, after special processing for upper and lower case, underlining, and literal blanks. This single line is printed independently of filling and justifying, or page-formatting processes; it is not counted as an output line.

Printing Selected Pages and Sections

The SELECT directive prints specified pages of a document. Like the OUTPUT directive, the SELECT directive is placed before any lines that are to be printed on the output device, perhaps in a header file. Although the entire text is formatted, only the selected pages are printed, saving unnecessary printing time.

The parameter consists of page numbers separated by spaces. Two page numbers separated by a colon selects the span of pages, including beginning and ending numbers. As shown below, the plus sign '+' specifies a second page number relative to the first. The next example prints pages 3, 5, 10 through 15 inclusive, and 20 through 25 inclusive.

```
.SELECT( 3 5 10:15 20:+5 )
```

By default, all pages are printed.

Appendix A: Summary Directive Table

Directive	Meaning (action)	Break	Parameter Type	Default
BREAK	Break justification	*	–none–	–none–
COMMENT	No action		remainder of line	–none–
COUNT	Set page count		numeric	.COU 1
FORM	Define page format	*	(...)	.FOR([/2 T #62 E /3 L54 + /3 #33 '-' PN:1 '-' /4])
INDENT	Indent next line	*	numeric	–none–
INPUT	Specify input options	*	(...) or numeric	.INP(D.W150 K+1)
INX	Store index entry		remainder of line	–none–
LITERAL	Print literal text		remainder of line	–none–
MARGIN	Set margins	*	(...) or numeric	.MAR(L0 R70)
OPTION	Set options	*	(...) or numeric	.OPT(S1 F+ H+ P+ L+ + R+ J3 E+ U- K+1)
OUTPUT	Specify output device		(...)	.OUT(LPT, SO U+)
PAGE	Eject to top of page	*	numeric	.PAG 5
PARAGRAPH	Set paragraphing params		(...) or numeric	–none–
RESET	Reset directive defaults	*	(...)	–none–
SELECT	Select pages to print	*	(...)	–none–
SKIP	Skip output lines	*	numeric	–none–
SORTINDEX	Sort and print index	*	(...)	.SOR(S1 L2 R2)
SUBTITLE	Set the subtitle		remainder of line	–none–
TITLE	Set the main title		remainder of line	–none–
UNDENT	Undent next line	*	numeric	–none–

The directives marked with an asterisk (*) cause a justification break before they are processed, since they affect the filling and justifying environment.

The ellipsis (...) indicates that the parameter is enclosed in parentheses and is described in detail along with the description of the directive itself.

Appendix B: Examples of PROSE Directives in Text

Example 1.

Input to PROSE:

```
.OUTPUT (LPT U+ E+)
.COMMENT  +---------------------------------------------------+
.COMMENT  |NOTE:  The header file supplied by Oregon Software  |
.COMMENT  |contains the output command for the line printer.   |
.COMMENT  |Do not use the above command if using that header.  |
.COMMENT  +---------------------------------------------------+
.FORM ([ // #10 Pn #28 T /// L50 //// ]
+      [ // #60 Pn #17 S /// L50 //// ])
.COMMENT  Page numbering starts at 62
.COUNT 62
.COMMENT  This combination of form and count directives
.COMMENT  duplicates the facing-pages format used in the
.COMMENT  many typeset books.
.COMMENT
.TITLE The Programmer
.SUBTITLE Chapter Three
.MARGIN(L10 R62)
.INPUT (N/ U_)
.PARAGRAPH (F& I2)
&Computers, Eddie knew, have no idea where their sources of
information are in the world. They look upon the world as one great
big fat wire bulging with information and instructions, a wire with
no beginning, middle, or end. The world for a computer is merely an
electrical input saying, ''Here's what you should know,'' or, ''Here's
what I want to know,'' or, ''Here's what you are to do now,'' and an
electrical output for them to talk back:
''Here's what I must know to answer your question,'' and, ''Here are
your answers.''  To the computers, all interrogators and commanders
speak with the same voice and the same authority; all listeners have
the same ear.
&Like guns. It doesn't matter to a gun who pulls its trigger. Guns
have awesome power, but they are entirely dependent on the hands that
use them. Morally guns and computers are out of it all, though they
are regularly the instruments for people who make things happen.
&Twice now--first with Betty's parking ticket and now with his
$25,624.34--Eddie had been someone who made things happen. It was
a very exciting sen/sa/tion, one he hadn't previously experienced.
&He had seen, not long before on the ''Today'' show, an airline pilot
who talked about his af/fec/tion for the 747. He loved it more, he said,
than any other aircraft he had ever flown, and he had been a pilot for
twenty-five years. The inter/view/er asked him to explain his
enthusiasm.
```

&''I sit there in that little room in the front,'' the pilot said, ''a room just four stories off the ground when we're parked, but at the top of the world when we're in flight--and I move litle knobs and dials. None of them takes more than a few ounces of pressure. In an instant a machine weigh/ing a hundred tons responds more smoothly than if I were moving it myself. It's like the aircraft becomes an extension of myself because so little effort is needed to make it do what I want, and it does whatever I want it to do. It's very ex/cit/ing.''
&''You make it sound almost sexual,'' the inter/view/er said.
&The pilot frowned. ''I don't know about that. I never thought about that.'' His face brightened. ''It's not sex. It's better. It's _real_ power.''
&Eddie sensed that his entire relationship with the computer had started a radical change. Before, he had been the machine's servant, bringing it little orders and loads of information to feed upon. The questions weren't his and the answers never mattered to him. He was merely an intermediary in the affairs of others. Now he was having his own affair.
&And his own affair required further action before the check in his pocket became anything but a useless piece of paper.
.SKIP 2
&On his way back to the office Eddie stopped in the motor vehicle section. Edna was there alone, as usual, talking on the telephone, also as usual. She was wearing a different pink sweater. She held up her hand, the fingers all extended. At first Eddie thought she was showing off her rings--there were four of them, all different--but then he understood that she was telling him she would be on the phone five minutes longer. He waved his hand to indicate he was in no hurry. She leaned back in the chair, her pink breasts pointing toward the corner light fixture.
&Eddie wandered around the office, acting as if he were bored. He looked at some papers. She was paying no attention to him. He stopped at the drawer where blank drivers' licenses were kept. He looked over his shoulder. She was still talking, her back to him. Her left hand held the telephone and her right hand slowly rubbed the back of her neck.
&He quickly took from the drawer ten forms, then leaned on the counter and quietly stamped each of them with the tricolored state seal required for validation. He put the forms into his jacket pocket along with the two checks. Now he had only to type out whatever names he wanted to use and he would have official New York certification.

Output from PROSE:

Computers, Eddie knew, have no idea where their sources of information are in the world. They look upon the world as one great big fat wire bulging with information and instructions, a wire with no beginning, middle, or end. The world for a computer is merely an electrical input saying, ''Here's what you should know,'' or, ''Here's what I want to know,'' or, ''Here's what you are to do now,'' and an electrical output for them to talk back: ''Here's what I must know to answer your question,'' and, ''Here are your answers.'' To the computers, all interrogators and commanders speak with the same voice and the same authority; all listeners have the same ear.

Like guns. It doesn't matter to a gun who pulls its trigger. Guns have awesome power, but they are entirely dependent on the hands that use them. Morally guns and computers are out of it all, though they are regularly the instruments for people who make things happen.

Twice now--first with Betty's parking ticket and now with his $25,624.34--Eddie had been someone who made things happen. It was a very exciting sensation, one he hadn't previously experienced.

He had seen, not long before on the ''Today'' show, an airline pilot who talked about his affection for the 747. He loved it more, he said, than any other aircraft he had ever flown, and he had been a pilot for twenty-five years. The interviewer asked him to explain his enthusiasm.

''I sit there in that little room in the front,'' the pilot said, ''a room just four stories off the ground when we're parked, but at the top of the world when we're in flight--and I move litle knobs and dials. None of them takes more than a few ounces of pressure. In an instant a machine weighing a hundred tons responds more smoothly than if I were moving it myself. It's like the aircraft becomes an extension of myself because so little effort is needed to make it do what I want, and it does whatever I want it to do. It's very exciting.''

''You make it sound almost sexual,'' the interviewer said.

The pilot frowned. ''I don't know about that. I never thought about that.'' His face brightened. ''It's not sex. It's better. It's real power.''

Eddie sensed that his entire relationship with the computer had started a radical change. Before, he had been the machine's servant, bringing it little orders and loads of information to feed upon. The questions weren't his and the answers never mattered to him. He was merely an intermediary in the affairs of others. Now he was having his own affair.

And his own affair required further action before the check in his pocket became anything but a useless piece of paper.

On his way back to the office Eddie stopped in the motor vehicle section. Edna was there alone, as usual, talking on the telephone, also as usual. She was wearing a different pink sweater. She held up her hand, the fingers all extended. At first Eddie thought she was showing off her rings--there were four of them, all different--but then he understood that she was telling him she would be on the phone five minutes longer. He waved his hand to indicate he was in no hurry. She leaned back in the chair, her pink breasts pointing toward the corner light fixture.

Eddie wandered around the office, acting as if he were bored. He looked at some papers. She was paying no attention to him. He stopped at the drawer where blank drivers' licenses were kept. He looked over his shoulder. She was still talking, her back to him. Her left hand held the telephone and her right hand slowly rubbed the back of her neck.

He quickly took from the drawer ten forms, then leaned on the counter and quietly stamped each of them with the tricolored state seal required for validation. He put the forms into his jacket pocket along with the two checks. Now he had only to type out whatever names he wanted to use and he would have official New York certification.

Example 2.

Input to PROSE:

```
.COMMENT  Output directive is in the header file.
.INPUT( B# N/ )
.OPTION( K1 )
.FORM( [ // #50 N #60 E /// L50 / #30 '- ' PN:1 ' -' ] )
.MARGIN( K1 L5 R70 )
.PARAGRAPH( K1 F& S1 )
&Something clicked in another part of his mind and he knew he was about
to become a portable computerized superpower.
&The question had been puzzling him for some time. It had to do with
pro/gram access. If one had a pro/gram--if one knew the para/dig/matic
structure of a set of encoded information--then one could do nearly
any/thing one wanted with that information. If it was simply material
stored, then one could learn everything that was stored; if it was
operational material, then one could command the operations. The
problem was, one needed the program to do the work, and the utility
of the programs he had taken with him when he left Buffalo was
limited.
&The cold water swirled around his legs and the ripples moved out from
where his hands paddled the surface. Suddenly it was as if the answers
had typed themselves out on the console screen.
.OPT( K2 U+ S2 )
.MAR( K2 L15 )
.PARAGRAPH( K2 F& U8 S1 )
&QUESTION:##HOW DO I FIND OUT WHAT PROGRAMS EXIST
WHEN I DON'T KNOW WHAT QUESTIONS TO ASK?
&ANSWER:##ASK THE COMPUTERS WHAT QUESTIONS THEY CAN
ANSWER FOR YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.
.OPT.MAR.PAR
&He ran home through the woods without even bothering to
dry off. Mosquitoes pecked at his face. He dressed quickly,
hooked up the van, and sat down at his keyboard. He addressed
IFFI, the central law enforce/ment computer in Baltimore. The
acronym stood for Information#Filed#for#Future#Investigations.
He asked IFFI a question that translated as, ''What discrete
sets of information have you on hand and what are the access codes
for them?'' He set the machine for a printout rather than a
readout on the monitor.
&In seconds the Selectric began typing away. It typed for a long
time. Office Selectrics can handle about thirty characters a
second, faster than any human can go, but the ones built for
information processing went three times as fast. Nearly
one thousand five-/character units of information a minute, and
the machine seemed to be typing faster than he had ever seen it
go before...
```

Output from PROSE:

3:35 PM 14 Jun 82

Something clicked in another part of his mind and he knew he was about to become a portable computerized superpower.

The question had been puzzling him for some time. It had to do with program access. If one had a program--if one knew the paradigmatic structure of a set of encoded information--then one could do nearly anything one wanted with that information. If it was simply material stored, then one could learn everything that was stored; if it was operational material, then one could command the operations. The problem was, one needed the program to do the work, and the utility of the programs he had taken with him when he left Buffalo was limited.

The cold water swirled around his legs and the ripples moved out from where his hands paddled the surface. Suddenly it was as if the answers had typed themselves out on the console screen.

QUESTION: HOW DO I FIND OUT WHAT PROGRAMS EXIST WHEN I DON'T KNOW WHAT QUESTIONS TO ASK?

ANSWER: ASK THE COMPUTERS WHAT QUESTIONS THEY CAN ANSWER FOR YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.

He ran home through the woods without even bothering to dry off. Mosquitoes pecked at his face. He dressed quickly, hooked up the van, and sat down at his keyboard. He addressed IFFI, the central law enforcement computer in Baltimore. The acronym stood for Information Filed for Future Investigations. He asked IFFI a question that translated as, ''What discrete sets of information have you on hand and what are the access codes for them?'' He set the machine for a printout rather than a readout on the monitor.

In seconds the Selectric began typing away. It typed for a long time. Office Selectrics can handle about thirty characters a second, faster than any human can go, but the ones built for information processing went three times as fast. Nearly one thousand five-character units of information a minute, and the machine seemed to be typing faster than he had ever seen it go before...

Appendix C: Historical Notes

Most text-formatting programs available today descend from one of several original programs. Among these is RUNOFF, developed on the Dartmouth Time-Sharing System in the 1960s. Later, the Call-a-Computer system provided a RUNOFF version called EDIT RUNOFF as a text-editor command. In 1972, Michael Huck, working on the University of Minnesota's MERITSS system (a CDC 6400 running the KRONOS operating system), began to develop a version of EDIT RUNOFF called TYPESET.

TYPESET was intended as a "versatile text information processor commonly used to typeset theme papers, term papers, essays, letters, reports, external documentation . . . , and almost any other typewritten text."* In spite of these aspirations, no program can be all things to all people. TYPESET went through many changes, stablizing somewhat in early 1977 at version 5.0, which is written in CDC COMPASS assembly language. John P. Strait developed PROSE, written in Pascal, over a year's time starting in the spring of 1977. The design was influenced heavily by TYPESET, making PROSE one of the many descendants of RUNOFF. *

PROSE, with minor changes, was installed on the Univac 1100 series computers in early 1980 by Michael S. Ball of the Naval Ocean Systems Center. At Oregon Software, he converted this version from the Univac to the PDP-11 in July 1980 and to the Motorola MC68000 in the spring of 1982.

* Michael Huck, *Typeset 5.0 Information*, © 1977.

For More Information

We suggest several places to find more information about Pascal and the environment in which Oregon Software's products are used. Many of the books are available from Oregon Software. Prices are subject to change without notice.

Algorithms + Data Structures = Programs Niklaus Wirth. Prentice-Hall. A study of programming data structures, beginning with the fundamental structures such as records, arrays, and sets and progressing to those structures that are changed in value and structure by program execution. Full-length sample programs illustrate the stepwise refinements involved in developing Pascal programs.

Elements of Programming Style Brian Kernighan, P. J. Plauger. McGraw-Hill. A practical demonstration of the principles of good programming and the use of common sense. The authors critique and rewrite programs from various texts on programming.

Introduction to Pascal Rodnay Zaks. SYBEX Inc. A complete tutorial on Pascal designed to be read and understood by everyone.

Oh! Pascal Doug Cooper, Mike Clancy. W. W. Norton. An easy-to-read Pascal course for the novice programmer.

Oregon Pascal Users Society (OPUS) An organization dedicated to the sharing of information between Oregon Software and its customers. Oregon Software is not affiliated with OPUS but encourages its activities and provides space for an OPUS column in its *Pascal Newsletter*. OPUS membership is free. OPUS is temporarily without a coordinator and address. You may send inquiries to Oregon Software, C/O OPUS COORDINATOR, and we'll forward your letter to the society.

Pascal Newsletter Oregon Software. Published several times a year, Oregon Software's *Pascal Newsletter* contains status reports on all of our Pascal products, announcements of new versions of software and new products, and various technical articles.

Pascal Standard, International The international Pascal standard *ISO 7185* is identical to the British standard, which is available from ANSI, prepaid. Request document *BS 6192* and write: ANSI, International Department, 1430 Broadway, New York, NY 10018.

Pascal Newsletter, The Published by the Pascal Users' Group, *The Pascal Newsletter* is available on a subscription basis. Contact:

Pascal Users' Group
2903 Huntington Rd.
Cleveland, Ohio 44120.

Pascal User Manual and Report Second Edition. Kathleen Jensen, Niklaus Wirth. Springer-Verlag. The first definition of standard Pascal.

Programming in Pascal Peter Grogono. Addison-Wesley. A good course in standard Pascal, with lots of sample programs for experimentation. (Oregon Software supplies one copy to each new customer.)

Standard Pascal User Reference Manual Doug Cooper. W. W. Norton. Provides a correct, comprehensive, and comprehensible reference for Pascal; an alternative to the ISO standard for "students and implementors with merely human powers of understanding."

Structured Programming O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare. Academic Press. Three monographs on methodologies of concept modeling (Dijkstra), data structuring (Hoare), and structured, hierarchical programming (Dahl and Hoare).

Systematic Programming: An Introduction Niklaus Wirth. Prentice-Hall. A description of the technique of constructing and formulating algorithms in a systematic manner, intended as general mathematical background rather than as a practical study of coding.

Index

Style note: Page numbers in boldface indicate the defining use of the term.