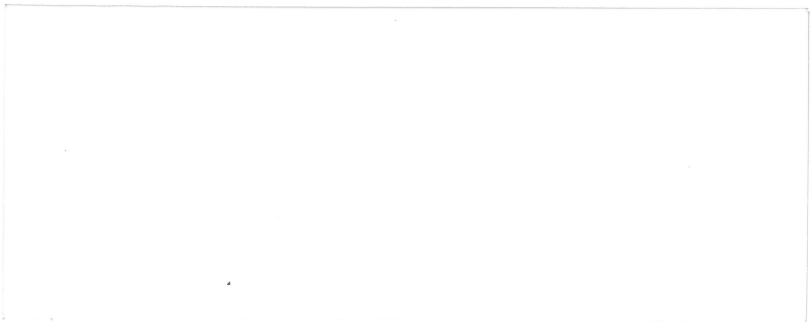
Pascal-2™

Version 2.1 for RSX-11

Oregon Software



Pascal-2®

Version 2.1 for RSX

Abbreviated Manual

July 1985

Oregon Software

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

List of Figures

Preface

We are grateful to the many readers whose thoughtful and perceptive suggestions have helped us improve our manuals. Please feel free to tell us your opinion of this abbreviated manual; we find both general comments and specific criticisms to be helpful. Address your comments directly to:

David Spencer, Manager
Technical Publications
Oregon Software
6915 SW Macadam Ave.
Portland, Oregon 97219

We appreciate hearing from you.

Thanks to...

Oregon Software got its start at OMSI—the Oregon Museum of Science and Industry. OMSI is a private educational organization chartered "to enhance the general public understanding of science and technology, with a strong commitment to education," particularly of youth. It was in the Research Laboratory at OMSI that we began writing software. Seven of us came from OMSI to found Oregon Software in September, 1977. Because of the close association, the name "OMSI" stayed with us for a while, and we continue to support OMSI and its educational programs.

As part of our own research, we've been using and distributing TEX, a text-processing system developed by Don Knuth at Stanford University. This publication is typeset in the Computer Modern Roman family of type faces with the TEX system. Draft versions were produced at Oregon Software on an Imprint-10 laser printer driven by TEX-in-Pascal and our VAX-11/780.

Special thanks to Barbara Beeton and Monty Nichols for their encouragement and assistance with TEX and to David Kellerman and Barry Smith for developing TEX at Oregon Software.

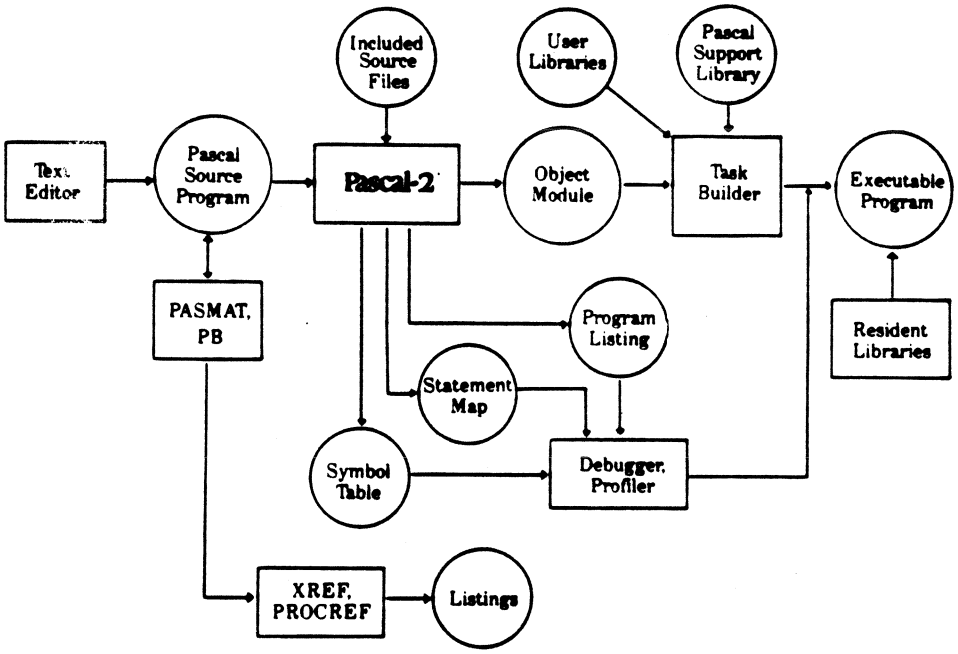
Pascal–2 V2.1 for RSX: System Features

This abbreviated manual summarizes the use of Pascal–2 on PDP-11 running under the RSX operating system. Included are an overview of the Pascal–2 development environment, a description of the complete documentation package, and a System Guide that walks you through the basic steps involved in running the Pascal–2 compiler.

The Pascal–2 Software Development System

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.

The Pascal–2 compiler accepts standard Pascal code created using any common text editor (EDT). All actions of the compiler are invoked by basic command lines and switches, similar in format to those of the Digital Command Language (DCL). The compiler permits calls to external routines both in Pascal and other languages such as FORTRAN and the DEC assembly language MACRO-11. The object code generated by the Pascal–2 compiler is linked with external and support library routines by the DEC Task Builder.



The Pascal–2 Software Development Environment

The Pascal–2 system consists of the 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, not shown, is also a component of the system. The user creates the Pascal source program, the included source files, and user libraries. The Text Editor and Task Builder are supplied by the computer vendor.

Together, the components and features of the Pascal–2 software development system offer professional programmers a structured and unified environment in which to design, code, test, maintain,

and improve software. As a result, programmers should be able to produce more reliable programs in less time than they could with other programming packages. Further, use of the Pascal–2 compiler allows programs to be transported to other computer systems with a minimum of change, thereby accelerating future software development.

Pascal–2 Documentation Package

Documentation for Pascal–2 Version 2.1 includes the *Pascal–2 User Manual for RSX*, Update Package No.2 for RSX, and the *Installation Guide/ Release Notes.*

The *Pascal–2 User Manual for RSX* contains information on the use of the Pascal–2 compiler and related utilities on Digital's RSX operating system. 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 Pascal language.

The user manual consists of five major guides:

- The User Guide serves as a quick overview of Pascal–2 on the RSX operating system. The guide, written on a beginner's level, takes you through the basic steps of compiling, correcting, and running a Pascal–2 program. The User's Guide also has brief explanations and examples of some of the standard features and utilities of the Pascal–2 system.
- The Programmer's Guide contains a detailed description of the low-level interface between Pascal–2 and the operating system. The Programmer's Guide also contains a miscellany of information on implementation-related problems, divided into two broad categories: error recovery and implementation notes. The guide further provides explanations of I/O switches that allow access to file system facilities. And finally, the guide 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 specification 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 that alleviate some tedious aspects of programming and improve the usefulness of the Pascal–2 system. The Debugger helps you to 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.

Update Package No. 2 for RSX, documents specific features of the 2.1D software and records changes made since the release of Update Package No.1. (Update Package No. 1 was previously distributed and all changes then noted have been incorporated into the manual.) Update Package No. 2 and provides a set of "change pages" for insertion into the user manual. (See "Documentation Notes" in the release notes before attempting to insert the change pages from the update package into the user manual.)

A set of release notes containing installation procedures accompanies each release of the software. These release notes should be read before attempting to install the compiler and utilities on your system. Each set of release notes should be kept with the manual for future reference because they

contain lists and descriptions of fixed bugs, new added features, and changes in the compiler and utilities.

Style Notes

This *Abbreviated Manual* and the *Pascal–2 User Manual for RSX* follow 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 the text are in boldface typewriter type. System directives are in upper-case boldface typewriter type: **WAITFR**, **SPAWN**. Program, system, and file names are in upper case: ROTAT, RSX.

Program Examples:

Commands, parameters and qualifiers that you should enter 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 fragment:

```
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 for RSX: System Guide

This System Guide is an extended version of the User Guide from the *Pascal–2 User Manual*. It explains how to compile and run Pascal–2 programs, how to interpret program listings and error messages, and it provides some details of the compilation process.

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

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 and store it in a file with the extension .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 the program—translate it into a form that the computer can execute. The Pascal–2 compilation process is directed by the **PAS** system command, which causes the Pascal–2 compiler to produce an object file with extension .OBJ. The Task Builder combines the object file with the Pascal–2 library to produce an executable program with extension .TSX.

To illustrate the compilation process, let's say that the following program is stored in the file FIRST.PAS.

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

Compilation and execution proceed as follows:

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

As the example shows, the .PAS, .OBJ, and .TSX extensions may be omitted from file names on commands to the Pascal–2 system, but such extensions must be included in commands to other RSX system programs such as the editor. Notice also that in this example, no errors were detected. The next example shows what happens if detectable errors are present in the source program.

Correcting Compilation Errors

Many times, a program contains syntax errors. The Pascal–2 compiler detects nearly 150 types of these "grammatical" errors in a program: errors 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:

```
>PAS SECOND
Pascal-2 RSX V2.1D   9-Feb-84   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)
                                  ^20
*** 20: 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 with an arrow indicating the approximate position of the error and a message describing the error. (The number "19" is the error message number generated by the compiler.)

The Program Listing

To correct an error, you often 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 on the command line, as shown in the following example. The listing also may be written to the line printer or by default to a disk file.

```
>PAS THIRD,TI: = THIRD/LIST
Pascal-2 RSX V2.1D   9-Feb-84   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)
                                  ^20
*** 20: 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. 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 previous example, a compilation switch modifies the compilation process in some way and is signified by a slash followed by a descriptive name. The next section describes all Pascal–2 compilation switches and their functions illustrated by samples of the compilation process.

Correcting 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 compiled and linked.

A run-time error such as "array subscript out of bounds" stops the program at the point of error. The Pascal–2 error reporting system prints header information and 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

TT0 -- Fatal error at user PC= 7244
Array subscript out of bounds

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

The first line of the walkback contains header information about the task and gives the task name or terminal port, the type of error (fatal or I/O) and the program counter at the time of the error.

The second line of the walkback is the error message issued by the error reporting system. (If the error is I/O-related, an additional line is printed that provides the system I/O error code and the name of the file causing the error.)

The third line shows the location of the error in terms of source program lines. The remaining lines of the walkback indicate the reverse order in which the procedures were called.

Pascal–2 also includes the ability to trap and recover from run-time I/O errors and to print additional information about the error.

A complete list of run-time error messages with explanations appears in Appendix B of the Programmer's Guide in the *Pascal–2 User Manual for RSX*. A more detailed explanation I/O error trapping and recovery also appears in the RSX Programmer's Guide in the section "Run-Time Error Recovery."

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*

where the components of the compilation line are supplied as follows:

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 are placed in the default directory. The default input file extension is .PAS. Multiple input files are separated by commas.

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

listing-file
: The listing file receives the compilation or error listing. The default listing file extension is .LST.

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 the invocation command or after any file specification. A switch applies to the entire compilation command regardless of whether it appears before or after the file specifications.

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 All **real** variables are in 8-byte floating-point format. You also must use colon notation (e.g., E:18:15) within the program to obtain double-precision values in the **write** statement. Default is "off": **real** variables are in 4-byte format.

pascal1 Specifies that the interface to external procedures be compatible with Pascal–1. This interface is a bit less efficient than that of Pascal–2; the **pascal1** switch 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.

novalkback
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. Default is **walkback**: the tables are generated, and the full walkback in source terms is displayed after the message header and error message. The **debug** compilation switch disables the generation of the error walkback.

Compiler Options

errors Requests that the listing file contain only lines with errors. Unless **list** is specified, the default is "on" and errors are printed on the terminal. When **list** is specified, the default is "off" and all lines are printed in the listing file. This switch 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, the default is "on"; otherwise the default is "off."

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. 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**.

test Used in debugging the compiler. Default is "off."

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 particulars of each switch are described in the following sections.

Program Options

$double Specifies that all real arithmetic is to be done with double precision rather than with single precision. **$double** applies to the entire compilation. You also 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."

$pascal1

Specifies that external procedures are called in a manner compatible with Pascal–1. This switch may slow program execution but should simplify conversion of programs from Pascal–1 to Pascal–2. The default is "off."

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** settiing applies to the entire compilation. Default is "off": global variables are shared.

$novalkback

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 procedure walkback is still issued, but addresses are substituted for line numbers and procedure names in the walkback diagnostics, saving considerable space. Default is **walkback**: the tables are generated, and the full walkback in source terms is displayed after the message header and error message. See "Run-Time Error Reporting" later in this guide for a discussion of the walkback. The **debug** compilation switch disables the generation of the error walkback.

Compiler Options

$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.

$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. 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. 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.

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

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

The Task Builder

The Task Builder combines the main program with routines from the Pascal and system libraries to produce an executable task with extension .TSK. 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 and the system library to produce 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 a disk as necessary and may be dynamically extended.

To include external modules, you may add their 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 also be added; they are marked with the **/LB** switch:

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

You may add a second output file to create a memory map which displays the contents of the task with the addresses and memory requirements of each component. The map file is created with the .MAP default extension.

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

Task Builder options make more resources available for large tasks. Two options commonly used with Pascal programs are **UNITS** and **EXTSCT**. The **UNITS** option increases the number of files available by increasing the logical unit numbers (LUNs). The LUN allocated determine the maximum number of files which may be open at any time. Six LUNs are allocated by default, two of them for **input** and **output**. When your program uses the Debugger or more than four files, you should allocate more LUNs with the **UNITS** option as shown below.

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

The **EXTSCT** (Extend Section) option allocates additional memory for a program section. **EXTSCT** parameters specify the section name and the number (in octal) of bytes of memory allocated 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

Using the Utilities

The programmer utility package is a set of procedures and routines designed to enhance the capabilities of the Pascal–2 compiler.

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 N = (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 source formatters. Utility routines such as PASMAT can be invoked by either a direct or indirect command line. In order to implement indirect commands, you need to create a command file in the directory [1,2]. This command file runs the utility program and defines an argument that allows you to pass input files to the routine. (See the section "Utility Invocation Commands" in the RSX Installation Guide for details.) In this example give the following direct command:

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

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.
```

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 you 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. First, compile and 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:

```
Pascal-2 RSX V2.1D   9-Feb-84   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.

```
>RUN EFACT

Pascal Debugger V3.00 -- 29-Nov-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
```

Double Precision

The computed value in the previous examples is printed with 7 significant digits. You may need greater precision for some programs. To get extended precision, use the **debug** switch, which computes to 15 significant digits. The **doubleswitch** allows you to print a more precise value for **e**.

```
>PAS EFACT/DOUBLE
>TKB EFACT/FP/CP=EFACT,LB:[1,1]PASLIB/LB
>RUN EFACT
With 19 terms, the value of e is 2.718281828459045
```

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.

Compile and task-build the program using 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>//
```

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

```
>RUN EFACT

profile  V2.1B  6-Feb-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 file looks like this:

```
Pascal-2 RSX V2.1D   9-Feb-84   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.

Index

Style note: Page numbers in boldface (**6**) indicate the defining use of the term.