Pascal-2™

Version 2.0 for RT-11

Oregon Software

Contents & Introduction

User's Guide

Programmer's Guide

Language Specification

Debugger & Profiler

Utilities

Conversion

Installation

Pascal-2®
Version 2.0 for RT-11

This publication is printed in the Computer Modern Roman family of type faces and has been typeset with the TEX typesetting system. Draft versions were produced at Oregon Software on a Versatec printer/plotter driven by TEX-in-Pascal and our VAX-11/780. Finished pages were produced at the American Mathematical Society, Providence, Rhode Island, on an Alphatype CRS phototypesetter driven by TEX in SAIL and a DECSystem-2060.

We wish to thank:

- Donald Knuth and the TEX group at Stanford University, for giving to the world this elegant and machine-independent tool;
- Barbara Beeton and the American Mathematical Society, for a cheerful voice in the wee morning hours, and for the impeccable print quality you see here;
- and Monte Nichols of Sandia Laboratories, for his personal encouragement and support of our typesetting experiments.

August 1982
Printed in USA

Contents & Introduction

Contents

Contents

Pascal–2 V2.0/RT-11 Introduction

Pascal–2 is a transportable multi-pass compiler that emphasizes conformance to the Pascal standard while generating optimized code. Properly used, **Pascal–2** will allow programs to be transported between computer systems with a minimum of change. The compiler itself has already been installed on computers made by two different manufacturers, and it is currently being implemented on two others.

Developed over several years, **Pascal–2** grew out of our experience with Oregon Software's first Pascal compiler, **Pascal–1**. **Pascal–1** is a one-pass compiler specific to the PDP-11 series, with low-level extensions giving the programmer control over the PDP-11 hardware and operating system. **Pascal–2** is larger and compiles more slowly than **Pascal–1**, but **Pascal–2** produces code that is much shorter and faster than **Pascal–1** code. Typical programs are 30 to 40 percent smaller and up to twice as fast.

This user manual contains all the information you need to install and operate the **Pascal–2** system on Digital's RT-11 operating systems: RT-11 V3 and V4, SJ, XM, and TSX-Plus by S & H Computing.

The first section is the User's Guide, which serves as a quick overview of **Pascal–2**, to give you a feel for how it works. The guide, which is 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 the Debugger, the Profiler, and the extended-precision format for `real` numbers — the standard features of the **Pascal–2** system.

The Programmer's Guide contains detailed descriptions of compilation commands, embedded and low-level switches, and the low-level interface between **Pascal–2** and the PDP-11 operating system. The Programmer's Guide also contains a miscellaneous collection of articles on implementation-related problems, divided into two broad categories: error situations (and what to do about them), and implementation notes. Finally, the guide describes **Pascal–2**'s optimizations, and it lists all of **Pascal–2**'s error messages.

The Language Specification describes **Pascal–2**'s language features in detail. Because not everyone is familiar with the major changes in the language since Jensen and Wirth's **User Manual and Report** in 1978, the Language Specification begins by summarizing those changes and describing the ways that **Pascal–2** deals with them. Thus, the guide serves not only as a description of our implementation of Pascal but also as a review of the language's evolution since 1978.

The Debugger Guide and the Utilities Guide describe programs designed to improve the usefulness of the **Pascal–2** system or to alleviate the tediousness of programming. The Debugger helps you find and correct errors that cannot be caught at compile time. The Utilities package contains program formatters, a text formatter, cross-reference programs, an execution profiler, a package that helps you to interface MACRO-11 routines with **Pascal–2** programs, and a dynamic string package. Each utility is described in detail, with examples.

The Conversion Guide, geared toward present **Pascal–1** users, 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 you isolate areas in your **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 section concludes with a list of solutions to errors that you may encounter while completing the conversion to **Pascal–2**.

The Installation Guide describes the steps required to install **Pascal–2** in your operating system.

This **Pascal–2** manual assumes that you have a basic familiarity with the Pascal language. Some sections, such as the Programmer's Guide and the Language Specification, assume a relatively detailed working knowledge of the language. Beginners can make their way carefully through this manual, but we encourage them to read the books described in the next section.

For More Information

We suggest several places to find more information about Pascal:

(1) Try it! Certainly the most challenging course, and the most open-ended and accurate as well. Acquire the habit of answering your questions by experiment. Remember, "You can't hurt the computer!"

(2) **Oh! Pascal**, by Doug Cooper and Mike Clancy — an easy-to-read Pascal course for the novice programmer.

(3) **Programming in Pascal**, by Peter Grogono — a good course in standard Pascal, with lots of sample programs for (1), above.

(4) **Pascal User Manual and Report**, by Kathleen Jensen and Niklaus Wirth — the first definition of standard Pascal.

(5) This manual — a description of the fine points and grubby details concerning **Pascal–2**.

For the serious student, these books are available from Oregon Software and elsewhere:

Algorithms + Data Structures = Programs, Niklaus Wirth; Prentice-Hall, $20.25

Structured Programming, Dahl, Dijkstra, Hoare; Academic Press, $15.30

Elements of Programming Style, Kernighan and Plauger; McGraw-Hill, $3.95

Systematic Programming: An Introduction, Niklaus Wirth; Prentice-Hall, $17.75

And we recommend that you join the Pascal Users' Group, which publishes an excellent newsletter. Send $10 for a one-year subscription to:

Pascal Users' Group
Attn: Rick Shaw
P.O. Box 888524
Atlanta, Georgia 30338
(404) 252-2600

Support Policy

The license fee for your **Pascal–2** system includes one year of software support, which covers the following:

1) Telephone assistance. We'll provide a quick cure to your problem if at all possible.

2) Formal, written response to all problems, suggestions, and comments received in writing. For complex problems, we need written descriptions of your technical problem to ensure correct diagnosis and repair. (This service does **not** include applications consultation.)

3) A no-cost update to the latest revision of **Pascal–2**, upon the written request of your Designated Contact Person. This is the standard response to bugs that have been fixed. (A handling fee is levied for some media; no charge for magtape and floppies.)

4) The Oregon Software Pascal Newsletter, which contains status reports on all of our Pascal products, announcements of new versions of software and new products, and various technical articles.

Support may be renewed annually.

Customers of an Oregon Software distributor will receive the Newsletter directly from us but should contact their distributor for other elements of support.

A Note on Style

This manual uses the following conventions:

Text: Sentence punctuation goes outside all quoted material. Pascal reserved words, predefined symbols and directives are in bold face typewriter type: **begin**, **write**. Program or system names are in upper case typewriter: ROTAT.PAS, PROFIL.OBJ. Commands are in typewriter: $fpp, /list.

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

Program Listings: **Pascal–2** accepts any combination of upper-case and lower-case characters; for consistency, the examples 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;
```

Who Are We, Anyway?

Oregon Software traces its origins to the real OMSI — the Oregon Museum of Science and Industry. OMSI is a private educational organization chartered "to further the education of the youth of the community", and 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.

But please, we're **Oregon Software**. We're a software research and development corporation in Portland, with a nice view of Mount Hood (and what's left of Mount St. Helens!) The seven from OMSI (the museum) have grown into twenty-five from all over.

On a serious note: OMSI is a non-profit, charitable institution. Contributions of money and equipment are much needed and are tax-deductible. The Research Lab supports independent science projects in many fields, including computing. For further information about the OMSI Research Lab program, contact:

Director of Research
Oregon Museum of Science and Industry
4015 SW Canyon Road
Portland, Oregon 97201
(503) 248-5943

And Finally . . .

Oregon Software plans to continually improve its written materials. Please send any suggestions in writing to:

Collins Hemingway
Documentation Editor
Oregon Software
2340 SW Canyon Road
Portland, Oregon 97201

User's Guide

Contents

Pascal–2 V2.0/RT-11 User's Guide

Introduction to the User's Guide

This is the introductory section, the User's Guide. It explains:

1) how to compile and run Pascal programs;
2) how to interpret program listings and error messages;
3) some details of the compilation process.

This guide assumes that you are familiar with:

1) simple RT-11 commands;
2) a text editor (e.g., EDIT, TECO, KED);
3) elementary Pascal programming.

This guide is **not**:

1) an introduction to Pascal (see **Programming in Pascal** by Peter Grogono);
2) a detailed description of **Pascal–2** (see the Language Specification, and Jensen and Wirth's **Pascal User Manual and Report**);
3) an expert's guide to **Pascal–2** (see the Programmer's Guide).

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 a program; store the program 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 directly — and link it to the **Pascal–2** support library. With the compiler and the support library on the system disk and with with a source file called TEST.PAS, the entire compilation process follows this example:

```
.R PASCAL
*TEST

.LINK TEST,SY:PASCAL
```

As the example shows, the .PAS extension may be omitted from file names on commands to the **Pascal–2** system but must be included in commands to other RT-11 systems such as the editor.

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:

```
.R PASCAL
*FIRST

LINK FIRST,SY:PASCAL
.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 will detect 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:

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

A semicolon is missing between the program heading and the reserved word **begin**. Semicolon errors (the most common errors made by beginning Pascal programmers) are **always** detected by the compiler:

```
.R PASCAL
*SECOND
Pascal-2 RT-11 SJ V2.0H   5-Jun-81   7:21 PM    Site #1-1    Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
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's Guide for a complete list of detectable compilation errors.)

Compilation Options

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 (.LST) file, include the /list switch in the compilation command line:

```
.R PASCAL
*SECOND/LIST
```

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

```
.R PASCAL
*THIRD, TT: = THIRD/LIST

Pascal-2 RT11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
THIRD,TT: = THIRD/LIST

         1       program Third (output)
                                       ^19
*** 19: Use ';' to separate statements
         2       begin
         3         writeln ('Things get hazy if you stare at them');
         4       end.

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

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 site identification (the facility name and site number). 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 above example of /list, a compilation switch modifies the compilation process in some way. A switch is signified by a slash and a descriptive name. The Programmer's Guide describes all of the compilation switches, but the next examples show the most commonly used ones. The examples also demonstrate some of the features of the **Pascal–2** package — the Debugger, the Profiler, the PASMAT formatter, and double-precision **real** number format.

The Formatter

Say you have a program, EX. It calculates an approximation of **e** (the base of the natural logarithms) by summing the series

$$1 + 1/1! + 1/2! + 1/3! + ... + 1/N!$$

until additional terms do not affect the approximation.

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

```
program Ex(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.
```

In the interest of readability, you decide to format the program with PASMAT, one of the **Pascal–2** utility programs. Give the following command:

```
.R PASMAT
*EX
```

and the program is reformatted to look like this:

```
program Ex(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 other formatting options. See the Utilities Guide for details.) Now proceed to compile the program.

```
.R PASCAL
*EX

.LINK EX,SY:PASCAL
.RUN EX
With 11 terms, the value of E is 2.718282000000000
```

The Debugger

Even after correcting any syntax errors caught by the compiler, you may still get unexpected results when the program runs. **Pascal–2**'s interactive Debugger can help uncover and correct the problems in this situation. 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. (In most cases, you probably also will want to overlay the Debugger module. See the Debugger Guide for details.)

First, compile the program with the command:

```
.R PASCAL
*EX/DEBUG

.LINK EX,SY:PASCAL
```

The /debug compilation produces four output files: EX.LST, EX.SYM, EX.SMP, and EX.OBJ. You will need the EX.LST 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 the EX/DEBUG command, you will find it handy to have a printout of the EX.LST file. The file will look like this:

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
EX/DEBUG

 Line Stmt
    1      program Ex(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 is labeled Stmt and 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 breakpoints.

In the program EX, 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 will prompt for the name of the program and then allow 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.)

```
.RUN EX

Pascal Debugger V3.00 -- 27-Jan-81

Program name? EX
} 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 /double switch, which computes and displays 15 significant digits. (See the Programmer's Guide for details.)

```
.R PASCAL
*EX/DOUBLE

.LINK EX,SY:PASCAL
.RUN EX
With 19 terms, the value of E is 2.718281828459050
```

The Profiler

Finally, let's "profile" the program by using the /profile switch and by adding the PRFILE module to the Linker input. The Profiler will take control of your program and ask for the program's name. Next, the Profiler will ask for the name of the profile output file. The default extension is .PRO.

```
.R PASCAL
*EX/PROFILE

.LINK EX,SY:PRFILE,SY:PASCAL
.RUN EX

profile - V3.1  15-Mar-81

Program name? EX
Profile output file name? EX  ———————— Output goes to EX.PRO

With 11 terms, the value of e is 2.718282000000000

Program terminated.

Profile being generated
```

The output file, EX.PRO, will look like this:

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
        EX/PROFILE

          Line Stmt
            1      program Ex(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.

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's Guide, the Language Specification, and the Debugger Guide. And, if you presently use Oregon Software's **Pascal–1**, consult the Conversion Guide to determine the best ways to change over programs to **Pascal–2**.

Programmer's Guide

Contents

Pascal–2 V2.0/RT-11 Programmer's Guide

Introduction

The Programmer's Guide contains nitty-gritty information about **Pascal–2** for programmers well-versed in the Pascal language. This guide describes compiler commands, compilation and embedded switches, I/O control switches, and **Pascal–2**'s low-level interaction with the PDP-11. This guide also describes ways to handle common Pascal-related implementation questions on RT-11 and contains other miscellaneous information.

This guide is **not**:

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

Compiler Commands

Compilation of a **Pascal–2** program begins with the `R PASCAL` command. The **Pascal–2** compiler responds with an asterisk (*) as a prompt. You must then supply the file specifications and compilation switches.

Examples in this document list the compilation switches after the last file specification, but switches may appear after any file specification and will apply to the entire compilation.

File Specifications

The only required file specification is the input file. Multiple input files are concatenated in the order in which they are given, 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. The default file extension for any input file is `.PAS`. 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 output file specification consists of the output file and the listing 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 is in `MACRO` and the default extension is `.MAC`. The listing file specifies the file to receive the compilation or error listing, with a default extension of `.LST`.

An output file specification exists whenever an equal sign '=' appears in the command line. If an equal sign '=' appears but no file name is listed in the position of the output file, no output file will be generated. If no file name is listed in the position of the listing, a listing output will be produced only if errors exist; if errors exist, output will be to the user's terminal with the `/errors` switch assumed.

Compilation Switches

Compilation switches provide control over the files generated and over some aspects of the generated code. A switch is signified by a slash '/' and 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) will always identify a **Pascal–2** compilation switch (e.g., /che; /noche, /mac, /nomac).

Some switches, such as /object and /macro, are incompatible and will cause the error message "Conflicting switches specified" if used in the same compilation.

Formal Compilation Syntax

compilation-spec = [*output-spec*] *input-spec* .

input-spec = *input-file* {"," *input-file* }.

output-spec = [*output-file*] ["," [*listing-file*]] "=" .

input-file = *file-spec* .

output-file = *file-spec* .

listing-file = *file-spec* .

file-spec = <*any legal file specification*> {"/" ["no"] *switch* }.

switch = *identifier* .

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. (See the Extended Precision section for more details.)

/nomain: No main program is expected; only procedures are compiled. The switch is used to generate external procedures. If a main program is found, an error message is generated saying that an extra statement has 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 not shared with other programs or external routines. Default is "off": global variables are shared.

/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.

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.

/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: Enables generation of code to interface to the **Pascal–2** Debugger. Default is "off". The switch cannot be used with the /profile switch.

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

Code Switches

/object: Generates an object format output file with default extension .OBJ. Default is "on" unless an output specification is provided but no output file is listed. The switch cannot be used with the /macro switch.

/macro: Generates MACRO 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.

Processor Switches

The processor switch defaults to the processor option for the machine on which the compiler is running. Change the value by specifying exactly 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 will be done with calls to a floating-point simulator.

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

Checking Switches

/nocheck: Disables all run-time checks, including index range checks, subrange assignment check, nil pointer checks, stack checks, and case label checks. Note that compile-time errors are still detected. Thus, if /nocheck is specified, `var A:array [2..10] of integer; A[1] := 0;` will still be detected as an error, but `I := 1; A[I] := 0;` will not be. Default is /check.

/standard: Requests that all **Pascal–2** extended language features be flagged as errors. Default is /nostandard.

/test: Used in debugging of the compiler. Default is "off".

/times: Prints wall-clock time consumed by the compiler. Default is "off".

Compilation Examples

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

Example 1.

```
.R PASCAL
*PROG/LIST
```

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

Example 2.

```
.R PASCAL
*PROG,PROG=PROG
```

Equivalent to Example 1.

Example 3.

```
.R PASCAL
*PROG=PROG/NOCHECK/FIS
```

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

Example 4.

```
.R PASCAL
*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 option is assumed to be on, and code will be generated for the hardware options of the machine on which the program is being compiled.

Example 5.

```
.R PASCAL
* ,TT:=PROG
```

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

Linking and Executing

After compilation, **Pascal–2** programs must be linked to the support library before being executed. For Version 4 of RT-11, the sequence is:

```
.LINK PROG,SY:PASCAL
.RUN PROG
```

For Version 3 of RT-11, the sequence is:

```
.R LINK
*PROG = PROG,SY:PASCAL
*^C
.RUN PROG
```

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}
```

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

```
{$ind} or {$noi}
```

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. The $double switch is an example of this. In some cases, the "no" form of the switch is the one you would normally use, as with $nomain.

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

```
{$nodebug}
procedure P;
  begin
  {procedure}
  end;
{$debug}
```

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".

$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 an extra statement has been found. Default is $main: a main program is being compiled.

$own specifies that global-level variables will be unique to this compilation unit and will not be shared with other compilation units. $own applies to the entire compilation. Default is $noown, indicating that global variables will be shared.

$pascal1 specifies that external procedures will be 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 switch must be turned on whenever a **Pascal–1** procedure or function is called; $nopascal1 turns off the switch. The default is $nopascal1.

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

Compiler Options

$nodebug, $debug disables/enables some of the overhead of the **Pascal–2** Debugger. These two switches will have an 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 can be used to turn off some of the debugging overhead for procedures or functions that have already been fully tested. $debug can be used to restore debugging for other blocks of code.

$noprofile, $profile disables/enables some of the overhead of the **Pascal–2** Profiler. These two switches will have an effect only when the /profile compilation switch is specified. The

/profile switch generates the extra files needed for profiling and sets $profile "on". $noprofile can be used to turn off profiling for procedures or functions that do not need to be profiled, and $profile can be used to restore profiling for other blocks of code.

The state of the $debug/$nodebug and $profile/$noprofile switches when the **begin** of a block is compiled will determine debugging or profiling for that entire block.

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 will display the $nolist/$list switches, and the line numbers will 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
{program continues}
```

Lines with errors will be 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 will be the message "No such statement in this procedure". Other errors may also be produced.

$standard, like the compilation switch /standard, alters the language checking to mark all extended language features of **Pascal–2** 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 will then check the rest of the program for non-standard features, so that you can minimize your use of extensions. The $nostandard switch will be 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 will turn off all run-time checks. The four embedded switches listed below will cancel particular checks selectively. Any of the four can be placed at the start of the program to turn off a particular kind of check throughout. Or, "on/off" pairings can be used on a line-by-line basis within the program.

Turning off run-time checks will reduce 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 the 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 the generation of code that checks for **nil** pointer values. Default is $pointercheck.

$norangecheck cancels the subrange assignment check capability. No assignment to a variable of subrange type is checked as to whether the assigned value is within the allowed range. 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.

I/O Control Switches

The **reset** and **rewrite** standard procedures accept additional arguments specifying a file name of an external file, and a default name with default fields of the file name. These arguments can also include I/O control switches, which give explicit control of the operating system interface details.

The I/O switches appear in the file name or default name parameters as in this example:

```
rewrite(F,'data.dat/seek/span/size:12.');
```

A special device (TI:) also may appear in the **reset** and **rewrite** calls. The TI: device connects to the **Pascal–2** terminal driver and is used in place of the TT: driver for interactive use.

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters.

/buffersize:*n*	Allocate N bytes for buffer
/go	Programmed error handling
/odt	Single-character terminal input
/nfs	Non-File-Structured access
/seek	Direct-access file
/size:*n*	File storage allocation
/span	Records span block boundaries
/temp	Temporary file

/BUFFERSIZE:n Switch

Pascal–2 normally allocates the minimum space required for a file buffer, which is usually 512 bytes but is dependent on device and file characteristics. More efficient I/O transfers can be performed at the cost of additional memory. The /Buffersize:n switch specifies the storage to be allocated to a file buffer. The size value is a decimal number if terminated with a period, otherwise octal.

/GO Switch

I/O transfer errors are normally fatal and cause immediate program termination. The /Go switch indicates that transfer errors on the specified file are non-fatal and allow program execution to continue. In using this switch, the programmer accepts responsibility for checking the RT-11 I/O status code after each I/O operation. The error code for the previous I/O transfer error is available in the byte at address 52B.

/ODT Switch

The /ODT switch derives its name from the ODT Debugger (Octal Debugging Technique), which is driven by single-character commands. The /ODT switch is used with keyboard files, and indicates that each character read from the file is to be processed immediately without any wait for a carriage return or other action character. The /ODT switch also disables the normal one-character buffer effect of the **read** standard procedure.

Note that the /ODT is only effective for files on the TI: terminal device. Also, the rubout and Control-U keyboard editing capabilities are not effective with /ODT.

/NFS Switch

The /NFS switch is used to achieve non-file-structured access to a device that is normally file-structured, such as a disk device. Because direct access to such a device can destroy its directory structure, **Pascal–2** prevents non-file-structured access unless the /NFS switch is used.

/SEEK Switch

The /Seek switch enables the use of the direct-access **seek** procedure, and it permits both read and write access to the file variable so that records may be updated.

/SIZE:n Switch

The /Size switch used in the **rewrite** procedure specifies the space to be allocated for the file. The size of the file is given in blocks of 512 bytes, and is a decimal number if terminated by a period, otherwise octal.

/SPAN Switch

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 switch packs records more efficiently, with records spanning from one disk block to the next. This requires additional buffer memory, which is automatically allocated, and some additional computation. Spanned and blocked files are not generally compatible. Files created with /Span should be read with the same switch.

/TEMP Switch

This switch is used in **rewrite** to indicate a temporary file that will be deleted on termination. No file name is needed if this switch appears.

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 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 will generate 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.

External Modules

An external module is a program fragment containing at least one procedure, function, or main program. External modules are compiled independently and combined at link time. External modules may be combined into libraries to simplify the handling of common routines. (See the section on the Linker, Overlays, and the Librarian.) The external module interface allows inclusion of modules written in other languages, such as FORTRAN and MACRO.

The **external** directive declares a procedure or function as "external", which means that the procedure may be referenced by other modules or defined in another module. External routines can be declared only at the outermost level of a program, but they may be called from inner levels as with any other global routine.

The **external** directive is similar to the **forward** directive in that the declaration of an external routine is separated from the body of that routine. The body of the external routine can appear in only one compilation unit. Declarations may appear in several compilation units; in fact, an external declaration must appear in each compilation unit that calls an external routine.

For example, a **Pascal–2** external procedure is defined by:

```
procedure Proc(Arg: Argtype);  ---------- this is the declaration
  external;

procedure Proc;  ---------------------------------- this is the body
begin
  :
end;
```

External modules may reference global (static) variables, which are shared by all of the modules composing 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.

The **NonPascal** directive replaces **external** in references to external routines written in FORTRAN or MACRO. The **NonPascal** directive causes the generation of a PDP-11 standard calling sequence (the Pascal calling sequence places parameters on the stack, while the FORTRAN sequence points internal register `R5` to a list of parameters).

Each external procedure name consists of the first six characters of the procedure or function identifer and must uniquely identify the external routine, because the names are used as global entry points by the Linker. The title of the module containing the external procedure(s) consists of the first six characters of the output file name. Any underscore character '_' in a procedure name or module title will be converted to a period '.'.

Duplication of external symbols will cause the Linker error "`?LINK-W-Multiple definition`". A reference to a missing external routine results in the "`?LINK-W-Undefined globals`" error message.

External Module Libraries

Suppose you want a library of procedures that can be referenced by any program. For a particular program, you will 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 form a single object module and cannot be individually loaded. If procedures `A`, `B`, and `C` are compiled together and placed in a library, any

reference to one of them will cause all three to be loaded. 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 will cause 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 write external declarations 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. Include that header file in the compilation. No external reference will be generated for any external procedure in the header file that is not referenced by the program, so only those modules actually used by each compilation unit will be loaded into the final image file (assuming that the library modules were themselves separately compiled).

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

Cautions

Observe one caution when using the **external** or **NonPascal** directive. Parameters to external routines cannot be checked by the compiler for type conformance, so an accidental type mismatch will cause unpredictable results. Also, the compiler cannot verify the conformance of global data.

All external modules must be compiled with the same **real** precision (single or double) as the main program.

Anyone attempting to combine **Pascal–1** and **Pascal–2** external procedures should first consult the Conversion Guide.

The Linker, Overlays, and the Librarian

The Linker

Object modules produced by the **Pascal–2** compiler are compatible with object modules produced by the MACRO assembler, FORTRAN compiler, and other RT-11 system utility programs. The Linker, LINK.SAV, can produce overlaid executable programs, allowing much larger programs. The Librarian, LIBR.SAV, can build libraries of object modules. Some highlights of Linker and Librarian capabilities are covered here. See the RT-11 System User's Guide for details.

To run the Linker, give the command:

```
.R LINK
*
```

('*' indicates that the Linker is waiting for a command.)

The first command line can include the output file, a map file if desired, and up to six input files. The file PASCAL.OBJ, which contains the Pascal run-time library, must be used when linking a Pascal program. The example below links the object module MAIN with two others, SUB1 and LIB1, to produce a MAIN.SAV file and a MAIN.MAP file.

```
*MAIN,MAIN=MAIN,SUB1,LIB1/C
*SY:PASCAL
*^C
```

Overlays

The /O:N switch selects the overlay facilities of the Linker, where the parameter N indicates the overlay region number. Sets of modules allocated to the same region will be overlaid against other modules in the same region, with only one set of modules per region actually in memory at any one time.

The following sequence links a main program and several external modules into an overlaid executable file. The main program and the Pascal library are not overlaid and must be in the root segment (on the first command line). FIRSTA and FIRSTB do not call each other and are overlaid against each other in region 1. TWOA and TWOB do not call each other and are overlaid against each other in region 2. The set of NEXTA, NEXT2A, and NEXT3A are overlaid against the set of NEXTB, NEXT2B, and NEXT3B in region 3. No module in one set calls a module in the other set that is overlaid in region 3. The continue switch (/C) allows the input file list on the next line to be included in the linking.

```
.R LINK
*PROG = MAIN,SY:PASCAL/T/C
*FIRSTA/O:1/C
*FIRSTB/O:1/C
*TWOA/O:2/C
*TWOB/O:2/C
*NEXTA,NEXT2A,NEXT3A/O:3/C
*NEXTB,NEXT2B,NEXT3B/O:3
Transfer symbol? $START
*^C
```

Overlays for the **Pascal–2** Debugger are described in the Debugger Guide of this manual.

The Librarian

The Librarian combines relocatable object module files to form an object module library. This library may be included as input to the Linker, which will select only those modules needed by the program being linked. Note that a module always consists of the entire set of procedures and functions from its compilation. Individual procedures cannot be selected from a module.

For example, the dynamic string package STRING.PAS, supplied as a **Pascal–2** utility, can be edited to form 9 files, with each file containing one procedure or function. The 9 files can then be compiled and combined into a library containing 9 modules, as follows.

```
.R LIBR
*STRING=LEN,CLEAR,READS,WRITES,CONC/C
*SEARCH,INSERT,DELETE,SUBS
*^C
```

Run-Time Organization

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 Linker to produce an executable program image. The use of multiple psects provides greater flexibility for the combination of individual procedures into a program.

The compiler generates these psects:

<blank> The instruction code for the compilation unit. The blank psects for all compilation units are concatenated; compiled code will not attempt to write to this psect.

CONSTS 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 will not attempt to write to this psect.

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 will be overlaid by the Linker so that only a single copy will exist in the final program. Compiled code will not attempt to write to this psect.

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.

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 will instead be named with the first six characters of the program name, allowing multiple global variable segments. Compiled code will write to this psect.

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

Psect Name	Attributes
<blank>	I,LCL
CONSTS	D,CON,LCL
TABLES	D,OVR,GBL
SHIFTS	I,OVR,GBL
GLOBAL	D,OVR,GBL

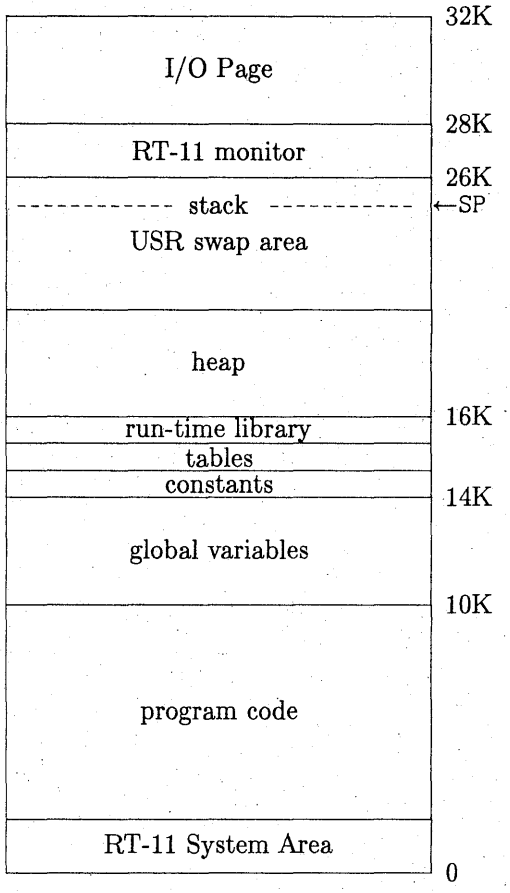
So that Pascal programs may be included in libraries, each **Pascal–2** object file has a module name consisting of the first six characters of the output file name. Thus a program compiled with the line:

```
.R PASCAL
*RESPROG = HDR,INPROG
```

will have the module name RESPRO.

Memory Organization

A program on the PDP-11 has access to 32768 words (frequently abbreviated to 32K). The exact arrangement of storage is determined by the commands to the Linker, but a typical program may look something like the following figure, which represents a snapshot taken during execution. The numbers are representative and will vary from program to program.



RT-11 System Area

The RT-11 System Area occupies the first 256 words of all programs and contains interrupt vectors and status indicators used by RT-11. This area is also used for communication between the Pascal program and other programs linked by chaining.

Program Code

The program code section contains the instructions for the user program. The size of this section is determined by the amount of 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.

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.

Tables

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

Run-Time Library

This section contains routines from the Pascal run-time library used by a program. Only those routines needed by a particular program are loaded here.

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 stack starts at the highest available address and expands downward, and the heap begins just after the program image and grows upward. This allows the maximum room for growth in both.

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 will eventually exceed the limits of the stack space and cause the error, "`Stack exceeded memory`".

The Heap

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

The heap is allocated from the bottom of available memory and can grow until it meets the stack, which is allocated at the top of available memory.

Space is returned to the heap when files are closed or when variables previously allocated with **new** are deallocated with **dispose**. Such space is then available for further heap allocation. The error message "`New() exceeded memory`" will result if no space is available to satisfy a request for heap storage.

The Monitor and I/O Page

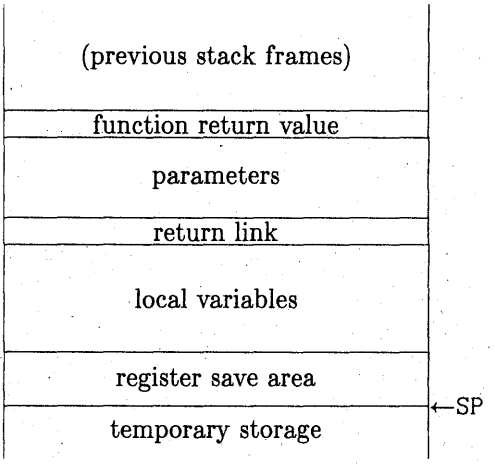
For RT-11 systems other than XM virtual jobs, the Monitor space contains the RT-11 resident monitor, the user service routine (USR) if it is set "NOSWAP", and any device handlers loaded with the `LOAD` command. The I/O page contains device status and command registers.

For an RT-11 XM virtual job (bit 10 set in the JSW), the monitor and I/O page are not allocated, and the stack will begin at the top of user memory. See the XM section for details.

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.

The format of a stack frame is:



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 will be stored in this location and left on the top of the stack when the function returns. Space for this field is allocated by the caller before evaluation of the arguments for the function call. The space will be removed from the stack when the caller 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 will contain the value of the corresponding argument, while the entry for a variable parameter will contain 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 will be the first one pushed onto the stack.

Return Link

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

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.) The register R5 is used as a pointer to the lexically enclosing block, and the compiler must be able to locate the value contained in R5 in each block to chain back to further 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.

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 Types: 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 Types: Allocated in the same way as the parent type.

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

Array Types: Allocated the amount of space needed to hold the number of elements specified, aligned in the same way as the element type.

Set Types: Allocated one bit for each member of the base type, with the total size rounded up to the next larger full 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 Types: 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 Types: 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 will contain 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. In this case, 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 Types: 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 Types: 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 will be forced to begin at a word boundary. Fields are allocated in the order declared, beginning at bit zero (least significant bit).

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. 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
```

An overflow condition in the main program will be 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. The easiest way to do this is to split the routine into several sub-procedures.

Consistency Checks

In addition to the above error messages, consistency checks within the compiler can (in theory) trigger one of these errors:

```
Undeleted temps in main program
Travrs build error in main program
Travrs walk error in main program
```

You should never see consistency-check errors; they are documented here for the sake of completeness. If you do see such an error, call Oregon Software immediately (503-226-7760) for a priority response.

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 will terminate with a "severe error" status. Otherwise, a "successful completion" status is returned.

The termination status can be used by the command file processor and the batch processor to terminate a command stream that encounters an error. For instance, a command file that compiles and links a Pascal program can use the compiler termination status to detect any errors and skip the link step.

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); external;
```

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

```
begin
  .
  .
  .
  Exitst(4);        { terminate with severe status }
  .
  .
  .
end.
```

A status of 1 means normal termination; a status of 4 means that an error terminated the program.

System Error Procedure

If a run-time error is detected, the system procedure Error is called with parameters describing the error and the system state. The Error procedure, known by the global name ERROR, may be replaced by a user-written external module of the same name. The external module must accept the parameters defined below.

```
type
  Class = (Fatal, IOError, Warning);
  Message = packed array [1..100] of char;

procedure Error(ErrorClass: Class;
                ErrorNumber: integer;
                ErrorMsgLength: integer;
                var ErrorMsg: Message;
                var XFile: text;
                IOStatus: integer;
                UserPC: integer;
                FilenameLength: integer;
                var Filename: Message);
```

The `ErrorClass` parameter indicates the type and severity of the error; `Fatal` and `IOError` are errors with no possible recovery, while `Warning` errors will recover automatically. The `ErrorNumber` indicates the exact cause of the error. See Appendix B for a list of values. `ErrorMsgLength` and `ErrorMsg` define the text of the printed error message normally displayed for this error. The `XFile` parameter identifies the file variable associated with this error, if any. `IOStatus` is the value of the I/O status word. `UserPC` is the program counter saved at this error, which can often be used to identify the program segment responsible for the error. Finally, `FilenameLength` and `Filename` describe the external name associated with the file variable `XFile`.

The possible courses of action available to the Error procedure are very limited, as a normal exit from the Error procedure results in program termination. The program global variables are available and may aid diagnosis of the problem. The Error procedure may provide operator interaction or recording capabilities beyond the normal messages to the terminal, and as a final resort may call upon operating system facilities to "chain" and restart the program or initiate another program.

Implementation Notes

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.

Your choice will depend on your needs. 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>;
```

The contents of the specified file are inserted at the point of the **%include** directive. If no file extension is specified, .PAS is assumed. The included file itself may contain **%include** directives, to a maximum nesting of seven levels.

The example below illustrates the use of both header files and the **%include** command.

Assume that the file CONFIG.PAS consists of this:

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

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

Assume also that the file COMDEF.PAS 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 file EXAMPL.PAS 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:

```
.R PASCAL
*CONFIG,EXAMPL
```

The result is an object module, EXAMPL.OBJ, that contains the output from the compilation of CONFIG.PAS, COMDEF.PAS, and EXAMPL.PAS, concatenated in that order. The object module can then be linked with the Linker to produce an executable image.

Any compilation switches used will apply to all input files.

Timestamp Procedure

The **Timestamp** procedure provides a way to get the date and time from within a Pascal program. Date and time will be obtained simultaneously so they will be consistent, even close to midnight.

Timestamp is included in the **Pascal–2** library, but the name is not pre-declared by the compiler. The user must include a definition similar to:

```
procedure timestamp(var day, month, year,              { date }
                    hour, min, sec: integer );         { time }
  external;
```

The following program will print the date and time using **Timestamp**.

```
program PrintTime(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
  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.
```

Foreground Operation

For foreground operation, allocate additional memory to ensure sufficient space for the stack, the heap, and file buffers. Each Pascal file requires about 300 words (more for large buffers), so allocate at least 600 words for the default input and output files. Use the /N: switch for V3 and the /BUFFER: switch for V4:

```
.FRUN <file name>/N:1024. {V3}

.FRUN <file name>/BUFFER:1024. {V4}
```

Virtual Jobs and the XM Monitor

The XM monitor has the ability to create a "virtual job" with a separate 32K word address space. A virtual job does not need to reserve space for the monitor, device handlers, or the I/O page, and can therefore use the entire available address space. A virtual job cannot make direct reference to

device registers or perform other specialized functions. These operations are restricted to "privileged jobs". To be compatibile with previous versions of RT-11, privileged jobs are the default type. For most purposes, however, virtual jobs are preferred.

Making a virtual job requires the setting of bit 10 (2000 octal) in the JSW (Job Status Word) in location 44 (octal) of the .SAV image file. Neither the Linker nor the Monitor supports this operation, but the VIRJOB.OBJ file supplied with **Pascal–2** will set the virtual bit in the JSW, for **Pascal–2** or other programs. VIRJOB should be included as an input file in the Linker command(s).

```
.LINK/XM TEST,VIRJOB,SY:PASCAL
```

One additional restriction: The .SAV image file containing a virtual job must be stored on the system device (SY:), and must be run via the R command.

Virtual overlays work with **Pascal–2** modules. For more information on virtual jobs and overlays, see Digital's RT-11 Software Support Manual.

Support Library

The Pascal support library is a collection of about 60 modules contained in an object module library called PASCAL.OBJ located on the system device. When compiling a program, the Pascal compiler generates subroutine calls to routines in the Pascal support library. The entry points in the library are identified as $Bxx where 'xx' is a small integer. To see these subroutine calls, inspect the MACRO code generated by the **Pascal–2** /macro switch.

Most of the routines in the Pascal support library deal with I/O operations or arithmetic computations such as floating-point simulation or trig function approximation. Other routines allocate dynamic memory and report error conditions. When you build a Pascal job, the Linker searches the Pascal support library for the modules required to run the job. For example, if you compute a logarithm in your program, the Linker will include the support library module that approximates logarithms (OPFLOG.OBJ).

In most Pascal jobs, the Linker includes from 2K words to 9K words of library modules.

The following table describes the modules in the Pascal support library PASCAL. The first column is the name of the module as it appears in the library. The second column is the name of the source file compiled to produce that module. The third column briefly describes the function of the module.

Module	Source	Description
DBG	DBG1 .PAS	Pascal–1 Debugger root module
DBGFID	DBGFID.PAS	Pascal–1 Debugger symbol file access
ERROR	ERROR .PAS	Error message printing
$APROX	OPAPRX.MAC	Double real rational approximation
$ARITH	OPINT .MAC	Integer multiply/divide/mod simulation
$CNVRT	OPCNV .MAC	Real/integer conversions
$DADD	OPDADD.MAC	Double real addition simulation
$DATN	OPDATN.MAC	Double real arc tangent
$DCMP	OPDCMP.MAC	Double real comparison
$DDIV	OPDDIV.MAC	Double real division simulation
$DEXP	OPDEXP.MAC	Double real exponentiation
$DLOG	OPDLOG.MAC	Double real logarithm
$DMUL	OPDMUL.MAC	Double real multiplication simulation
$DSQRT	OPDSQR.MAC	Double real square root
$DTIME	OPDTIM.MAC	Double time
$DTRIG	OPDTRG.MAC	Double sine/cosine
$DYNMM	OPDYN .MAC	Dynamic memory (new/dispose)
$ERROR	OPERR .MAC	Error processing

$FATN	OPFATN.MAC	Real arc tangent
$FCMP	OPFCMP.MAC	Real comparison
$FEXP	OPFEXP.MAC	Real exponentiation
$FLOG	OPFLOG.MAC	Real logarithm
$FLTIO	OPFLIO.MAC	Real number I/O common routines
$FORT	OPFORT.MAC	FORTRAN call interface
$FPSIM	OPSIM .MAC	Single-precision floating-point simulator
$FRACT	OPFRAC.MAC	Double-precision common routines
$FSIM	OPFSIM.MAC	Floating-point simulation trap linkage
$FSQRT	OPFSQR.MAC	Real square root
$FTRIG	OPFTRG.MAC	Real sine/cosine
$INIT	OPINIT.MAC	Initialize Pascal program
$IO	OPIO .MAC	Common I/O routines
$OPDBG	OPDBG .MAC	Pascal–1 Debugger trap interface
$OPEN	OPOPEN.MAC	File open
$P2DBG	OPDBG2.MAC	**Pascal–2** Debugger trap interface
$P2ERR	OP2ERR.MAC	**Pascal–2** error text
$PACK	OPPACK.MAC	Pack/Unpack
$READC	OPRDC .MAC	Read characters
$READI	OPRDI .MAC	Read integer value
$READR	OPRDR .MAC	Read real value
$READS	OPRDS .MAC	Read strings
$REG	OPREG .MAC	Register save/restore
$RESET	OPRSET.MAC	File Reset
$REWRI	OPRWRI.MAC	File Rewrite
$SEEK	OPSEEK.MAC	Seek record in file
$SET	OPSET .MAC	Set operators
$SWAP	OPSWAP.MAC	Swap real parameters on stack
$TIME	OPFTIM.MAC	Real time
$WRITC	OPWRC .MAC	Write characters
$WRITI	OPWRI .MAC	Write integer value
$WRITR	OPWRR .MAC	Write real value

Compiler Optimizations

The **Pascal–2** compiler implements these optimizations:

Variable Assignments to Registers

The compiler will permanently assign 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 used the most often. No register will be 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 will save 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. (The RT-11 SJ compiler does not fold **real** constants; the XM compiler does.) 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 will discard 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 will be generated if the identifier Debugging is defined as a constant with the value **false**.

Boolean Expression Optimization

When appropriate, **Pascal–2** will use 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 will be 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.)

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 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 will generate 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: Compiler Error Messages

```
'(' expected
')' expected
',' expected
'..' expected
':' expected
':=' expected
';' expected after procedure body
'=' expected
'[' expected
']' expected
']' or ',' must follow index expression
Actual parameter type doesn't match formal parameter type
Array exceeds addressable memory
Array subscript out of range
Assignment of file variables not allowed
Assignment operands are of differing or incompatible types
Assignment to constants not allowed
Assignment value out of range
BEGIN expected
Bad CASE label
Bad IN operands
Bad ORIGIN value
Bad constant
Bad parameter element
Bad type syntax
Badly formed expression
Binary operator expected
Block declarations are incorrectly ordered
Block ended incorrectly
Block must begin with LABEL, CONST, TYPE, VAR,
   PROCEDURE, FUNCTION, or BEGIN
Boolean value expected
CASE label does not match selection expression type
CASE selection expression must be a non-real scalar type
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 must be non-real scalar type
Case label type does not match tag field type
Compiler writer error -- please contact Oregon Software at (503) 226-7760
DO expected
Declaration terminated incorrectly
Declared labels must be defined in procedure body
END expected
Exponent must lie in range -38..38
Expression type is incompatible with FOR index type
External procedures/functions must be defined at outermost level
Extra END following block -- Check BEGIN ... END pairing
Extra procedures found after main program body
Extra statements found after end of program
```

```
FOR-loop control variable can only be a simple non-real scalar variable
Field variable expected for NEW
File cannot contain a file component
File names in RESET/REWRITE are non-standard
File variable expected
File variable or pointer variable expected
Files must be passed as VAR parameters
Format expression must be of type integer
Forward procedure/function body is never defined
Forward type reference is never resolved
Function cannot be applied to an operand of this type
Function identifier is never assigned a value
Function name expected
Function result must be of scalar or pointer type
Function result type cannot be duplicated in forward-declared function body
Identifier cannot be redefined or defined after use at this level
Identifier expected
Illegal character
Illegal comparison of record, array, file, or pointer values
Illegal function assignment
Illegal subrange
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
Label cannot be redefined at this level
Label defined twice
Label is target of illegal GOTO
Label must be declared in LABEL declaration
Label must be unsigned integer constant
Line too long
Must assign value before using variable
Need at least 1 digit after '.' or 'E'
Need at least one value to WRITE
Need at least one variable to READ
No strict inclusion of sets allowed
Non-standard comment form, please use "{" or "(*"
Nonsense discovered after program end
OF expected
OTHERWISE/ELSE clause in CASE not allowed
Octal constant contains an illegal digit
Octal constants are not standard Pascal
Only 15 levels of nesting allowed
Only functions can be called from expressions
Operand expected
Operands are of differing or incompatible type
Operator cannot be applied to these operand types
```

```
Packed array [1..n] of characters expected
Parameter list cannot be duplicated in
   forward-declared procedure/function body
Pointer variable expected
Procedure name expected
Procedures cannot be followed by type definition
Readln and writeln must be applied to text file
Reassignment of FOR-loop control variable not allowed
Record identifier expected
Set is constructed of incompatible types
Set types must have a base in the range 0..255
Sets must be non-real scalar type
Statement ended incorrectly
String constants may not include line separator
String of length zero
THEN expected
TO or DOWNTO expected
Tag does not appear in variant record label list
Tag identifier already used in this record
This function was declared as a forward procedure
This parameter cannot be followed by a format expression
This procedure was declared as a forward function
This procedure/function name has been previously declared forward
Too few actual parameters
Too many actual parameters
Too many errors!
Too many forward references (only 50 allowed)
Too many identifiers (only 1599 allowed)
Too many nested INCLUDE directives (only 8 allowed)
Too many procedures (only 250 allowed)
Too many strings or identifiers
Type name expected
UNTIL expected
Unary '+' or '-' cannot be applied to set operands
Undefined identifier
Unexpected ')' -- Check for matching parenthesis
Unexpected ELSE clause -- Check preceding IF for extra ';'
Unknown directive
Use '.' after main program body
Use ';' to separate declarations
Use ';' to separate statements
VAR parameters cannot be passed an expression or packed field
Variable name expected
Variable of type array expected
Variable of type record expected
Variables of this type are not allowed in READ
Variables of this type are not allowed in WRITE
Variant label is undefined
```

Appendix B: Run-Time Error Messages

```
 2 Array subscript out of bounds
32 Attempted reference through NIL pointer
18 Can't read
11 Can't reset(output)
11 Can't rewrite(input)
35 Case selector matches no label
30 Channel not open
36 Dispose() of Nil
 5 Division by zero
22 Duplicate Dispose()
17 End of file
 8 EXP overflow
30 File not open
33 File overflow
 6 Floating point format error
 3 Floating point overflow
29 FPP error
23 Illegal value for integer
20 Integer overflow
 9 LOG of zero or a negative number
 1 New() exceeded memory
21 New() of zero length
16 No channels available
19 Put not allowed
 3 Real overflow
30 Reset and no file
11 Reset failure
29 Reserved instruction trap
11 Rewrite failure
28 Seek on sequential file
27 Seek out of range
 1 Stack exceeded memory
 7 Square root of a negative number
33 Transfer error
29 Trap to 4
20 Trunc/round overflow
35 Variable subrange exceeded
33 Write past eof
```

Language Specification

Contents

Pascal–2 V2.0/RT-11 Language Specification

Introduction to the Language Specification

The **Pascal–2** compiler processes the standard Pascal language as described in the Pascal Revised Report by Niklaus Wirth, published by Springer-Verlag, corrected printing of 1978, and more completely described in the ISO Draft Proposal 7185.1, ISO/TC 97/SC 5 N595, dated January 1981. Compliance is Level 0: conformant array parameters are not included. **Pascal–2** includes the extensions detailed below.

This manual includes data on non-standard language features. This guide is not intended as a full language document. Differences between **Pascal–2** and **Pascal–1** are described in the Conversion Guide of this manual.

Examples of syntax definitions in this guide follow the protocol described in Appendix 3, **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 most recent draft of the standard (1981), this section outlines those changes and **Pascal–2**'s method of coping with them. Some of the features appeared in **Pascal–1** as non-standard elements when that compiler was developed and have since become standard. Other features are new to **Pascal–2**.

"For" Statement Controlled Variables

`For` statement controlled variables must be simple variables, local to the routine in which the `for` statement is written. Originally, any variable could be used.

File Declaration Point

The standard states that the files `input` and `output` are automatically declared as global variables if they are mentioned in the program heading. Since program headings are optional in **Pascal–2**, `input` and `output` are declared as global variables in every **Pascal–2** program. Thus, you cannot define a variable or procedure as `input` or `output` at the global level. In earlier versions of the language, the actual point of definition was undefined; in **Pascal–1**, `input` and `output` are defined at a level outside of the global level.

Parameter Compatibility

The compatibility rules for `var` parameters are now defined according to a restrictive rule called **name compatibility**. A variable passed as a `var` parameter must be declared with the exact type identifier used to declare the parameter. Previously, the rules for `var` parameters were undefined. (**Pascal–1** rules are less restrictive in places.)

Procedure and Function Parameters

The draft Pascal 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 changed to:

parameter-list = "(" *parameter-section* { ";" *parameter-section* } ")" .

parameter-section = (["var"] *identifier* {"," *identifier* }":" *identifier*) | *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
  P(P1);
end.
```

Strings

A string can no longer extend over more than a single line. The earlier standard was unclear on this point, and **Pascal–1** allows strings extending over more than one line. The change allows better diagnostics for unterminated strings.

"Write", "Writeln" of "Array of Char"

A **write** or **writeln** procedure call applied to an **array of char** will truncate the written string if the field-width parameter will not allow the entire string to be written.

Examples:

```
write(Buffer:BuffCount);    { write buffered characters }
write('cutoff':3);          { will write 'cut' }
```

Identifiers

Identifiers may be of any length; all characters are significant. Lower-case characters are interpreted in the same way as upper-case characters, so that **name**, **Name**, **NamE**, and **NAME** are equivalent. (**Pascal–2** also allows the **$** and _ (underbar) in identifiers. See Language Extensions for details.)

Alternate Symbol Representations

The standard now defines alternate representations for symbols that are unavailable in some character sets. These are:

Standard Symbol	Alternate Symbol
^ or ↑	@
{	(*
}	*)
[	(.
]	.)

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.

Examples:

```
(* This is a valid comment }
{This is (* not *) a valid comment }
```

Implementation Definitions

This section provides details and characteristics of implementation-defined elements of **Pascal–2**.

"Integer", "Real", "Char", "Text", and "Set" Types

The standard type **integer** has the range (-32768..32767). An unsigned (extended-range) integer may be defined with the range 0..65535.

The predefined identifier **maxint** has the value 32767.

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 gives 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"

Pascal–2 uses the 7-bit full ASCII character set. A character is stored as a byte unless it is part of a packed record. `Ord(char)` is in the range 0..127.

Standard Type "Text"

The standard type **text** is a file type with components of type **char**. **Text** is implemented as a file of 7-bit ASCII characters.

"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. The declaration **set of integer** is equivalent to the declaration **set of** `0..255`.

I/O Definitions

The following table summarizes the default field widths used when values are written to a text file:

Value Type	Field Width
integer	7
real	13
boolean	5

The floating-point representation of a real number includes an upper-case `E`, a sign character '+' or '-', and two exponent digits. For example, the real number —100.0 will print as `-1.000000E+02`. Positive numbers will have a blank in place of the negative sign.

Boolean values are written in upper case (**TRUE**, **FALSE**).

The procedure **page** is equivalent to `writeln(ff)`, where `ff` is the form-feed character.

`Reset(input)` performs the equivalent of a **readln**, but otherwise has no effect. `Rewrite(output)` will print any incomplete line, but otherwise has no effect. `Reset(output)` or `rewrite(input)` produces error messages. If the extended form of `Reset(input)` or `rewrite(output)` is used, the standard files input and output may be associated with external files in the same way as other file variables. For example, `rewrite(output,'LP:')` will switch the standard output to the line printer.

Syntax Extensions

This section describes extensions to the syntax of standard Pascal.

Identifiers

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. The following are legal identifiers:

```
system$name
$$file
this_is_a_long_identifier.
```

"%Include" Lexical Directive

A special directive form allows the inclusion of separate text files within a program. The syntax is:

include-directive = "`%include`" *file-name* ";"

The contents of the specified file are inserted at the point of the **%include** directive. If no file extension is specified, **.PAS** is assumed.

The included file may contain nested **%include** directives, to a maximum of seven levels.

Example:

```
%include hdr;
```

See the Programmer's Guide for more details.

Program Heading

The program heading and parameters are not required to be present in **Pascal–2**. If the program heading is present, it will be checked for proper form, but it has no other meaning to the program. An additional parameter to **reset** and **rewrite** specifies an external file. (See External File Access under I/O Support Extensions for details.)

Declaration Order

The declaration sections (**label**, **const**, **type**, **var**) may be interleaved as desired at the global level of a program. Only **const** and **type** may be interleaved at other levels. Any number of declaration sections may appear in any order. An identifier still must be defined before the identifier is used in any other way. This extension is useful for source module inclusion and structured constants as described below.

"External" and "NonPascal" Directives

The directive **external** may be used in a manner similar to **forward** to distinguish a particular Pascal procedure or function. The body of an external procedure or function is not required to appear in a compilation. An **external** procedure must be declared at the global level. References to the external procedure will be resolved at link time.

If the body of the external procedure does appear, its name will be made available in the object module for reference by other modules.

Note that 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 will be replaced by a period (.) in the external name. No type checking is done for parameters of an external routine.

The directive **nonpascal** 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 routines. This calling sequence passes all parameters by reference, so only **var** parameters may be used.

Structured Constants

The syntax for constant definitions is extended to allow the specification of constants of record or array type. The following changes are made to the syntax:

constant = [*sign*] (*unsigned-number* | *constant-identifier*)
| *character-string* | *structured-constant* .

structured-constant = *structured-type-identifier constant-component-list* .

constant-component-list = "(" *constant-component* {"," *constant-component* }")" .

constant-component = *constant* | *constant-component-list* .

The structured-type-identifier must name a type with an array or record structure, and all of the components of that structure must be of simple types or array or record types. The constant-components correspond one-to-one with the components of the structured type. Each constant-component must be a constant of the same type as the corresponding structure component. An access to the structure component will return the value of the constant-component. If the structure component is of a structured type, only the corresponding constant-component-list must be provided. The structured-type-identifier for that component need not be provided. For variant records (even those without a tag-field) a tag value must be provided in the constant-component-list.

Example:

```
type
  Compensation = (Paid, Unpaid);
  Paytype = Record
    Title : (Clerk, Indian, Chief, President);
    case Compensation of
      Paid:   (Rate: real);
      Unpaid: ();
    end;
  Employeetable = array[1..4] of record
    Name : packed array[1..10] of char;
    Payinfo : Paytype;
    end;
```

```
const
  Workers = Employeetable(
    ('Charlie   ', (Clerk, Paid, 3.40)),
    ('Samuel    ', (Indian, Paid, 5.25)),
    ('Maxine    ', (Chief, Paid, 6.85)),
    ('Edward    ', (President, Unpaid))
    );
```

Default Case Label ("Otherwise")

A default statement can be included in a **case** statement according to the following syntax:

case-statement = "case" *case-index* "of" *case-list-element* {";" *case-list-element* }[";"]
["otherwise" *statement* [";"]] "end" .

The statement after **otherwise** is the default statement and is executed if no case-constant matches the value of the case-index.

Example:

```
case I of
  1: Ch := ':';
  9: Ch := ';';
otherwise Ch := '*';
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** and **rewrite** accept an optional second parameter specifying the name of an external file with which the file variable is to be associated. The file name parameter must be a string type and may be either a literal string or a variable. An optional third parameter, also of a string type, provides default values for any file fields not provided in the file name.

An optional fourth parameter is set to the file size in blocks when the file is **reset** and specifies the number of blocks initially allocated on **rewrite**. The fourth parameter must be an integer variable. The fourth parameter, if present, is set to a status code of -1 if the file cannot be opened, allowing recovery from a normally fatal error.

These optional parameters may be used to redirect the standard files **input** or **output**, as in the example:

```
rewrite(output, outstring, '.lis', size)
```

"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. Note that files are automatically closed upon program termination, or when they appear in another **reset** or **rewrite**. This procedure allows the reuse of the buffer memory.

Direct-Access Files ("Seek")

Pascal–2 includes the **seek** predefined procedure to allow direct access (random access) to data files. The **seek** procedure requires two parameters: a file variable to be positioned, 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 variable 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. 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 }
```

Note that a **/seek** file switch must appear in calls to **reset** or **rewrite** if the **seek** procedure will be used on the file.

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 }
```

String Input ("Read" and "Readln")

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.

"Break" Procedure

For efficiency, **Pascal–2** buffers transmitted data. **Break**(F) forces the actual transmission of data from a partially filled buffer of file **F**. This can be useful with interactive terminals or to guarantee actual transmission of data to a shared disk file.

Real Number Formatting

If the second formatting field is negative, a **real** number will be printed in scientific notation. The number of digits to the right of the decimal point will be the number specified in the second field. For example,

```
write(R:20:-5)
```

will print 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 will be right-justified in a 20-character field.

Low-Level Interface

This section describes **Pascal–2** extensions that are useful to programmers needing access to machine-dependent PDP-11 characteristics.

Boolean Operators on Integer

The Boolean operators **and**, **or**, and **not** may be applied to operands of **integer** or **integer** subrange type. The operators produce a 16-bit result of **integer** type.

Octal Constants

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 —32768..32767, but you also may declare **integer** types in the range 0..65535. A variable with an upper limit greater than 32767 is called an **extended-range** variable. Normal arithmetic operations are performed on extended-range variables. Comparisons are unsigned. An integer value may be assigned to an extended value, being converted as a bit pattern. If the value being assigned is negative, the error is not trapped at run-time, since there is no way for the compiler to tell the difference between a negative value and an extended-range value. The same sort of implicit transformation is true when an extended value is assigned to an integer variable. No conversion is done for constants, and the assignment `bignumber := -1` will produce a compile-time error message if the variable `bignumber` is of an extended-range type. Values greater than 32767 will be read and written as negative numbers.

"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 will have 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 Function for more details on those.

The example controls a mythical device. The procedure `Read_Data` 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 machine, but the setting up of **origin** is the same for RSX or RSTS/E.

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
ORIGIN/LIST
    1       program Device;                 { example of device control }
    2
    3       const
    4         Ready = 200B;                 { ready flag }
    5         Read_Buffer = 1;              { read data command }
    6
    7       type
    8         Buffer = packed array [1..100] of char;
    9         Buffer_Pointer = ^Buffer;
   10
   11       var
   12         Status_Register  origin 177316B: integer;
   13         Control_Register origin 177314B: integer;
   14         Buffer_Address   origin 177312B: Buffer_Pointer;
   15         Byte_Count       origin 177310B: integer;
   16         Data: Buffer;                  { holds data from device }
   17
   18         procedure Read_Data;
   19         begin
   20           Buffer_Address := ref(Data);{ Address for DMA xfer }
   21           Byte_Count := size(Buffer); { size of buffer }
   22           Control_Register := Read_Buffer; { start transfer }
   23           { Wait for device to complete transfer }
   24           while (Status_Register and Ready) = 0 do {wait};
   25         end;
   26
   27       begin
   28         Read_Data;
   29       end.
```

"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). Note: 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 for use of **ref** in a program.

"Loophole" Function

The **loophole** function provides a controlled escape from Pascal type rules. The first parameter is a type identifier specifying the result type of the function. The second parameter is an expression of a 'compatible' type. In this context two types are considered compatible only if

1. They require the same amount of storage, or
2. They are both non-real scalar types.

For example, the following are all legal:

```
type
  Realequiv = array [0..1] of integer;
  scalar = (Va10, Va11, Va12, Va13);

var
  Re: Realequiv;
  R: real;
  S: scalar;
  I: integer;

Re := loophole(Realequiv, R);
R := loophole(Real, Re);
S := loophole(scalar, I);
I := loophole(integer, S); {equiv to I := ord(s);}
```

The result of the **loophole** function is the bit pattern of the argument, considered as a value of the type specified.

The only other method of type coercion (also non-standard) is to declare a record with variants, using the fact that the compiler overlays storage for different variants. The **loophole** function has several advantages over variant records:

1. No assumption need be made about field allocation in a variant record.
2. The compiler checks that the different types are the same size.
3. The bypassing of type checking rules is clearly marked (the compiler will flag **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 Dump;                          { dump memory contents }
type
  Word = 0..65535;

  procedure DumpMemory(Start, Finish: Word);
  type
    Pointer = ^integer;
  var
    P: Pointer;
  begin
    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;

begin                                  { main program }
  DumpMemory(1200B, 1240B);
end.
```

```
.RUN LOOPHOLE
  1200:    616
  1202:  16605
  1204:     10
  1206:  20566
  1210:      6
  1212: 101032
  1214:  10546
  1216:  12746
  1220: 177772
  1222:   4767
  1224:   1124
  1226:  12746
  1230:  11064
  1232:  12746
  1234:      2
  1236:  11646
  1240:   4767
```

"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, consider

```
type subrange = 0..15;
```

then

`size(subrange)` = 2, a full word is allocated,

and

`bitsize(subrange)` = 4, the bits required for the value.

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**.

Non-Standard Language Elements

Program Parameters

According to the standard, parameters supplied in the program header indicate external files. Further, the predefined files **input** and **output** must appear in the program header if they are used in the program.

With **Pascal–2**, the program header is not required, and any program parameters are entirely ignored. External files are referenced by an extended form of **reset** and **rewrite** using a second parameter (a string) giving the external filename.

The predefined files **input** and **output** are always defined at the global level, and may not be redeclared at that level.

Interactive Text I/O

The Pascal standard requires that the first element of a file be available as soon as the file is **reset** (the buffer variable **F^** is assigned a value immediately). This requirement can present serious difficulties when applied to files that are interactive terminals. For example, if the default input file is the user's terminal, the standard can be interpreted to require that the user first type the initial input character (or line) before the first program statement is executed.

Pascal–2 takes this route around the problem: When an interactive file is **reset**, the buffer variable is set to a space and **eoln(F)** is set to **false**, but no actual I/O transmission occurs. Each **read** request then waits for sufficient data to satisfy the request, but no more.

This approach solves most of the problems with interactive terminals in a predictable manner, but other difficulties may arise. When applied to an interactive file, the following program is unable to distinguish between an empty line and a line containing a single space. This is because **eoln** cannot be set until the end-of-line character is typed to satisfy the **read** request.

Example (the standard schema for reading a line of characters):

```
var Line: packed array[1..72] of char;
    Count: integer;
begin
  Count := 0;
  while not eoln do begin
    Count := Count+1;
    read(Line[Count]);
    end;
  readln;
end;
```

Directives

Pascal–2 does not allow an identifier to have the same name as one of the directives (**forward**, **external**, **nonpascal**). These directives are treated in the same way as reserved words.

"Eof" Not Accurate For Binary Files

An RT-11 file structure is a sequence of 512-byte blocks. A file containing short records may actually end in the middle of a block, but no information is available as to the end of valid data in the last block, so the **eof** standard function should not be relied upon as accurate. Another method, such as a sentinel record or a record count, should be used to indicate the end of usable data.

Eof is correctly indicated for **text** files.

"Mod" of Negative Numbers

The draft standard states that the operator **mod** always has a non-negative result. That is,

$$0 \leq I \bmod J < J.$$

Pascal–2 generates a PDP-11 divide instruction that will give a negative result if I is negative. The standard result can be generated by

```
if I < 0 then Result = I mod J + J
else Result = I mod J;
```

Additional Predefined Function "Time"

Pascal–2 provides the additional built-in function, **Time**. This function takes no parameters and returns a real value corresponding to the current time of day. **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 exact resolution of **Time** is dependent on the operating system, but all operating systems provide a resolution of at least one second.

Example:

```
procedure WriteTime;
var Hrs, Mins: integer;
    AmPm: packed array[1..2] of char;
begin
  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;
```

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. An attempt to call **get**, **read**, or **readln** when the file has not been **reset** or when **eof** is **true** for that file.
2. An attempt to call **put**, **write**, **writeln**, or **page** when the file has not been rewritten or when **eof** is **false** for that file.
3. **Ln** or **sqrt** has a negative argument.
4. The integer value returned by **trunc** or **round** lies outside the range **-maxint-1..maxint**.
5. Integer or real division by zero.
6. The result of a real operation cannot be expressed because of limitations in the floating-point format.
7. No label matches the value of the case index in a **case** statement.
8. The characters being read from a text file do not represent a legal value for the type of variable being read.

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 switch is 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 switch is enabled. (Default.)
3. A reference through a pointer with a **nil** or undefined value. Reference through a **nil** pointer is detected when the $pointercheck switch is 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 switch is enabled. (Default.)
5. An attempt to call **put** on a file that was opened with **reset**. Detected except for a file with the /seek switch specified when the file was opened.
6. 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.
7. The result of the **sqr** function is out of range. Detected if the argument type is **real**; undetected if the argument type is **integer**.
8. 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.
9. 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.
10. A **mod** with the right-hand side less than or equal to zero. Detected if the value is zero; otherwise not.
11. Reference to an undefined variable. Undetected in general. However, many simple cases are detected at compile time.

12. 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.

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. A call to **put** when the file variable is undefined.
6. A **reset** of a file that has not yet been written to. An empty file is made available for reading as the result of a non-standard language element in **Pascal–2**.
7. Accessing of a variable allocated with **new**$(p, c_1, ..., c_n)$ as an entire variable, in an assignment or as a parameter.
8. Calling of **dispose(p)** when the value of **p^** was created with **new**$(p, c_1, ..., c_n)$, or calling of **dispose**$(p, c_1, ..., c_n)$ with a variable created with **new** and a different set of tag values.
9. The result of an integer operation is incorrect because of overflow.
10. The value of a format expression to a **write** statement is less than 1. Undetected (used in a language extension).

Appendix 1: Predefined Identifiers

*** Extensions**

Constants

False
Maxint
True

Types

Boolean
Char
Integer
Real
Text

Variables

Input
Output

Functions

Abs
Arctan
Bitsize*
Chr
Cos
Eof
Eoln
Exp
Ln
Loophole*
Odd
Ord
Pred
Ref*
Round
Sin
Size*
Sqr
Sqrt
Succ
Time*
Trunc

Procedures

Break*
Close*
Dispose
Emt*
Get
New
Pack
Page
Put
Read
Readln
Reset
Rewrite
Seek*
Unpack
Write
Writeln

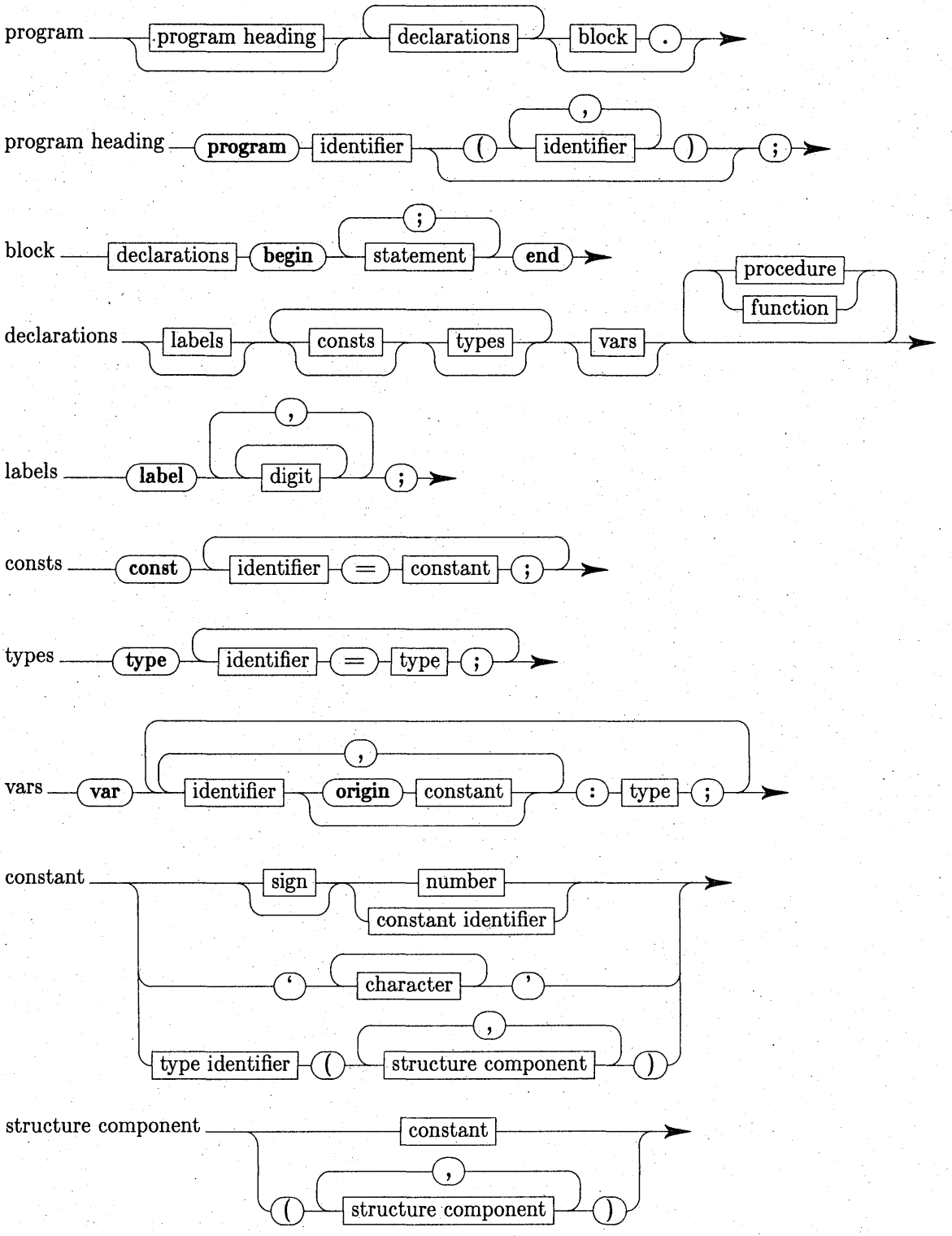
Appendix 2: Reserved Words

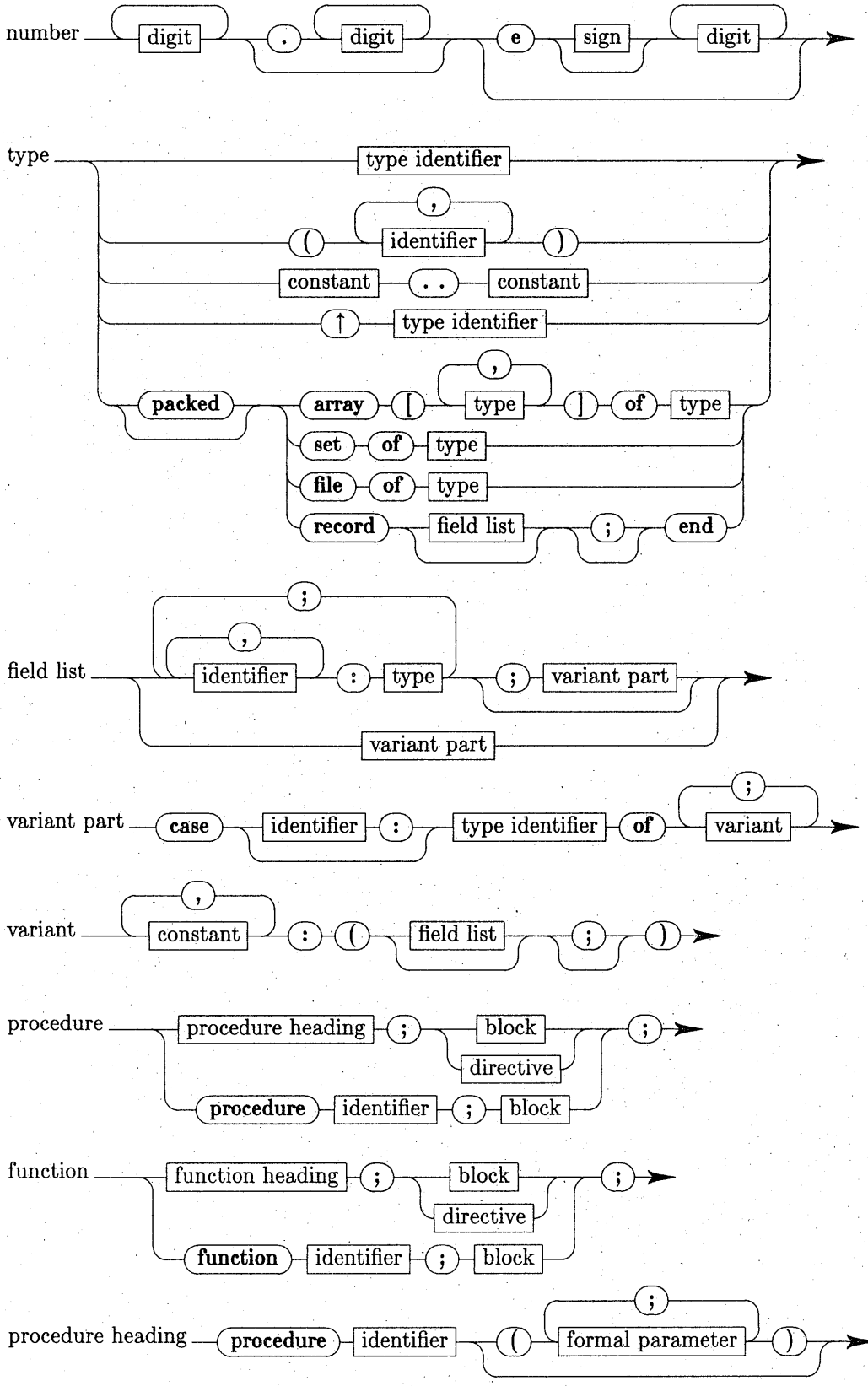
*** Extensions**

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
Origin*
Otherwise*
Packed
Procedure
Program
Record
Repeat
Set
Then
To
Type
Until
Var
With
While

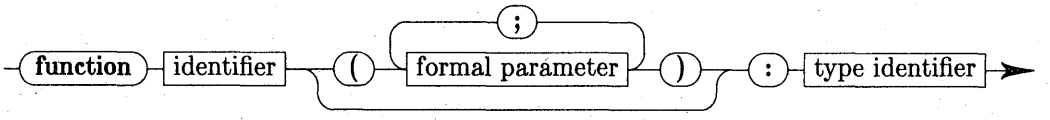
Appendix 3: Pascal–2 Syntax

Pascal–2 Syntax Diagrams

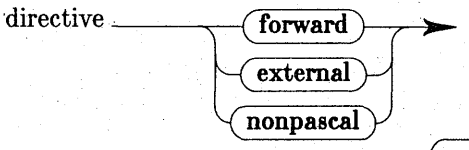



number
digit
.
digit
e
sign
digit
type
type identifier
(
,
identifier
)
constant
. .
constant
↑
type identifier
packed
array
[
,
type
]
of
type
set
of
type
file
of
type
record
field list
;
end
field list
;
,
identifier
:
type
;
variant part
variant part
variant part
case
identifier
:
type identifier
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
)

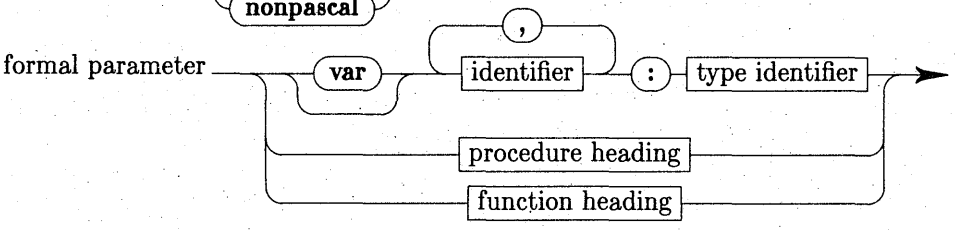
function heading



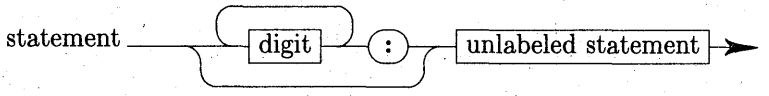
directive



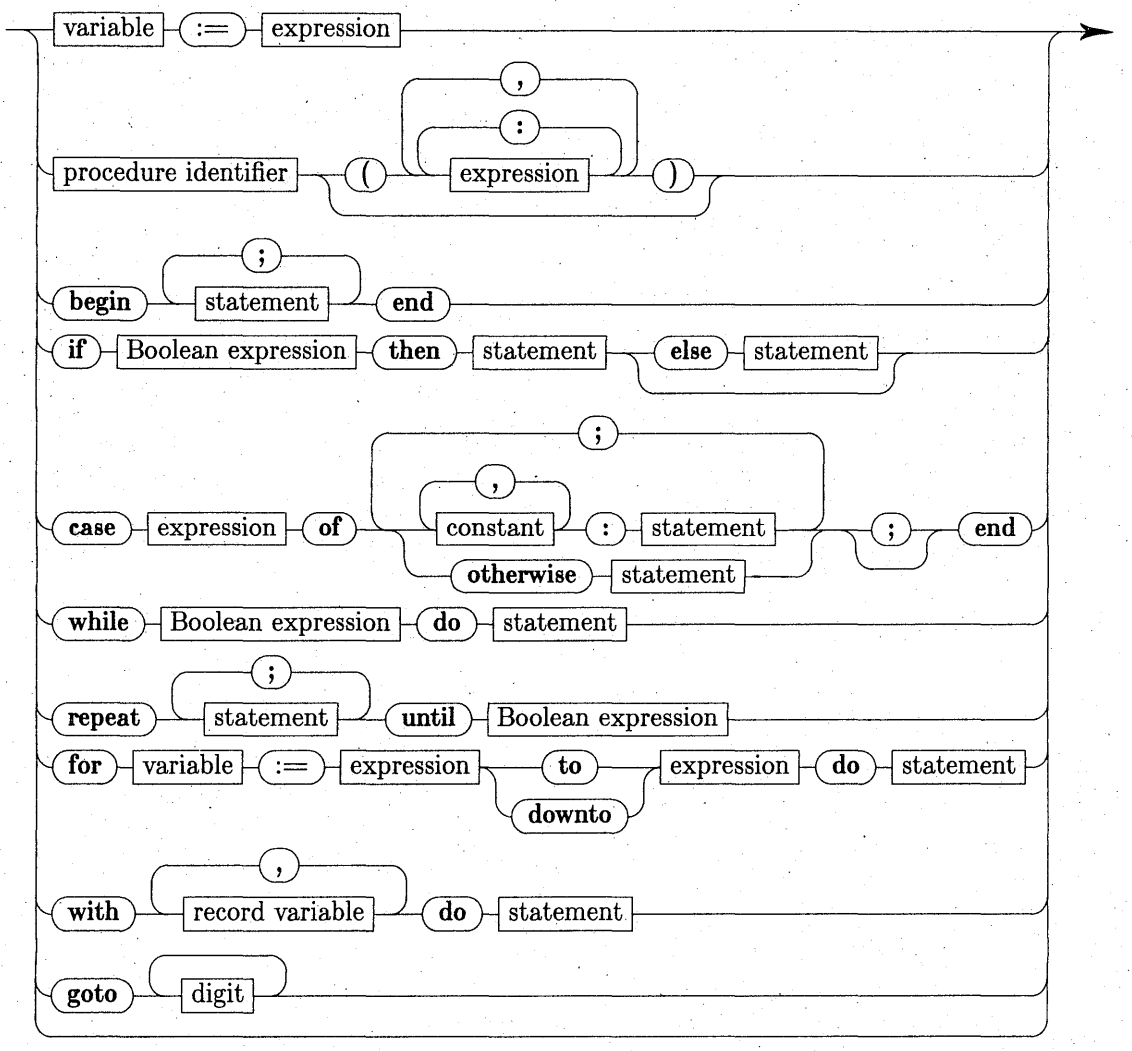
formal parameter

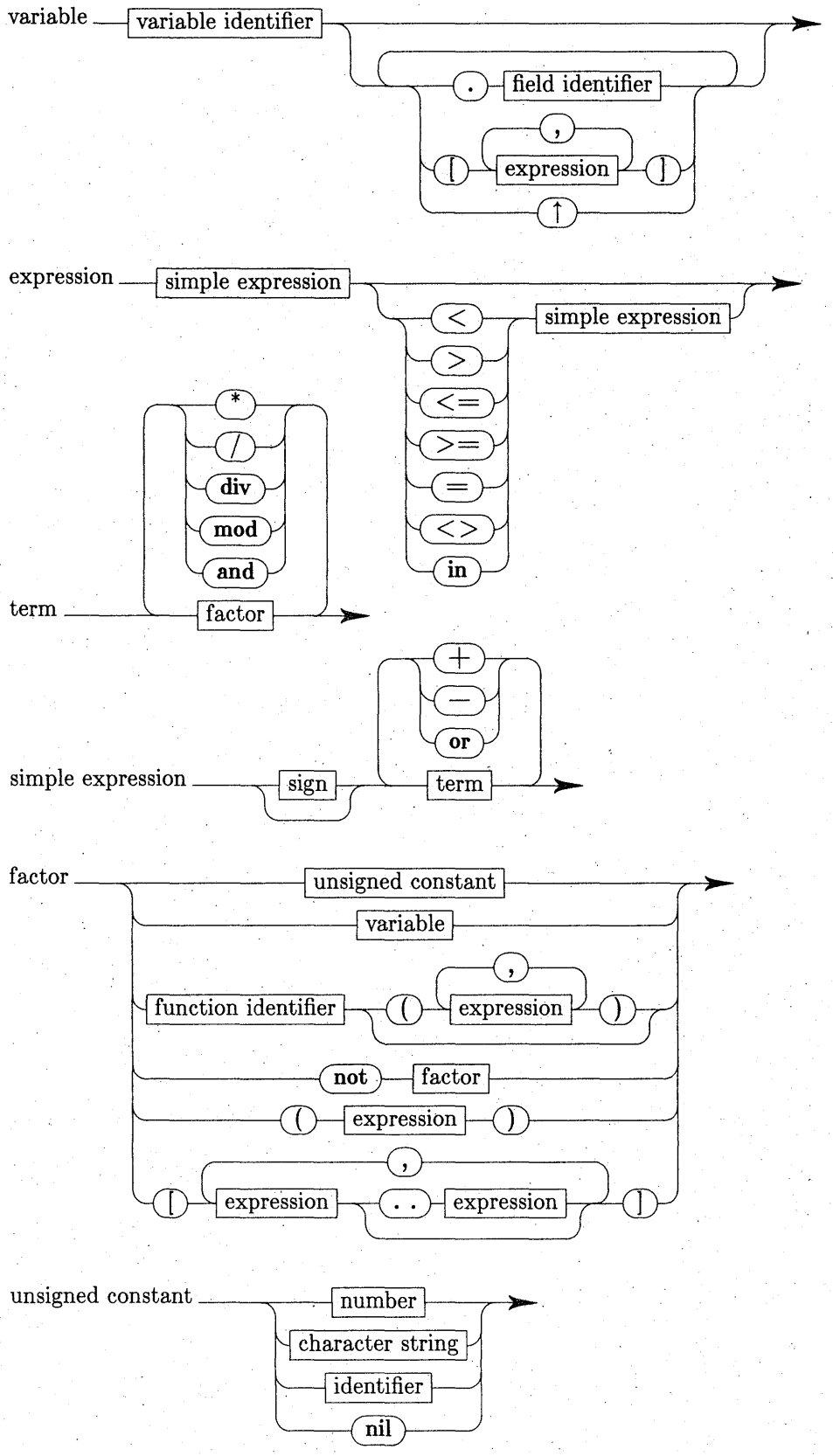


statement



unlabeled statement




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
function identifier
(
,
expression
)
not
factor
(
expression
)
[
,
expression
. .
expression
]
unsigned constant
number
character string
identifier
nil

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 will stand for some sequence of terminal or non-terminal 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 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 = *non-terminal-symbol* "=" *expression* "." .

expression = *term* {"|" *term* }.

term = *factor* {*factor* }.

factor = *non-terminal-symbol* | *terminal-symbol* | "(" *expression* ")"
| "[" *expression* "]" | "{" *expression* "}" .

terminal-symbol = "'" *character* {*character* }"'" | <*any comment in angle brackets*> .

non-terminal-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. *special-symbol* = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" | "(." | ".)"
| "." | "," | ":" | ";" | "^" | "@" | "(" | ")" | "<>" | "<=" | ">="
| ":=" | ".." | *word-symbol* .

5.* *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" .

6.* *identifier* = *letter* {*letter* | *digit* | "_" }.

7.* *directive* = "forward" | "external" | "nonpascal" .

8. *digit-sequence* = *digit* {*digit* }.

9. *unsigned-integer* = *digit-sequence;*

10. *unsigned-real* = (*unsigned-integer* "." *digit-sequence* ["e" *scale-factor*])
| (*unsigned-integer* "e" *scale-factor*) .

11.* *octal-number* = *octal-digit* {*octal-digit* }"b" .

12.* *unsigned-number* = *unsigned-integer* | *unsigned-real* | *octal-number* .

13. *scale-factor* = *signed-integer* .

14. *sign* = "+" | "–" .

15. *signed-integer* = [*sign*] *unsigned-integer;*

16. *signed-real* = [*sign*] *unsigned-real;*

17.* *signed-number* = *signed-integer* | *signed-real* | [*sign*] *octal-number* .

18. *label* = *unsigned-integer;*

19. *character-string* = "'" *string-element* {*string-element* }"'" .

20. *string-element* = "''" | <*any printable ASCII character*> .

21. *comment* = ("{" | "(*")
<*any sequence of characters and ends of lines not containing* "}" *or* "*)" >
("}" | "*)") .

22.* *lexical-directive* = "%include" <*host file name*> ";" .

Pascal–2 EBNF 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 ISO draft Pascal 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* {"," *indentifier* }.

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* = *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*) |
    procedure-heading | *function-heading* .
39. *body* = *compound-statement* .
40. *statement* = [*label* ":"]
    [*assignment* | *procedure-call* | *compound-statement*
    | *if-statement* | *case-statement* | *while-statement* | *repeat-statement* | *for-statemen*

    | *with-statement* | *goto-statement*] .
41. *assignment* = *variable* ":=" *expression* .
42. *procedure-call* = *identifier* [*arg-list* | *write-arg-list*] .
43. *arg-list* = "(" *expression* {"," *expression* }")" .
44. *write-arg-list* = "(" *write-arg* {"," *write-arg* }")" .
45. *write-arg* = *expression* [":" *expression* [":" *expression*]] .
46. *compound-statement* = "begin" *statement* {";" *statement* }"end" .

47. *if-statement* = "`if`" *expression* "`then`" *statement* ["`else`" *statement*] .

48.* *case-statement* = "`case`" *expression* "`of`" *case-element* {";" *case-element* }[";"]
["`otherwise`" *statement* [";"]] "`end`" .

49. *case-element* = *constant* {"," *constant* }":" *statement* .

50. *while-statement* = "`while`" *expression* "`do`" *statement* .

51. *repeat-statement* = "`repeat`" *statement* {";" *statement* }"`until`" *expression* .

52. *for-statement* = "`for`" *variable* ":=" *expression* ("`to`" | "`downto`") *expression*
"`do`" *statement* .

53. *with-statement* = "`with`" *expression* {"," *expression* }"`do`" *statement* .

54. *goto-statement* = "`goto`" *label* .

55. *expression* = *simple-expression* [*relational-operator simple-expression*] .

56. *relational-operator* = "<" | ">" | "<=" | ">=" | "=" | "<>" | "`in`" .

57. *simple-expression* = [*sign*] *term* {*adding-operator term* }.

58. *adding-operator* = "+" | "-" | "`or`" .

59. *term* = *factor* {*multiplying-operator factor* }.

60. *multiplying-operator* = "*" | "/" | "`div`" | "`mod`" | "`and`" .

61. *factor* = *unsigned-constant* | *variable* | *function-call*
| ("`not`" *factor*) | ("(" *expression* ")")
| (("[" | "(.") [*member-designator*
{"," *member-designator* }] ("]" | ".)")) .

62. *unsigned-constant* = *unsigned-number* | *string* | *identifier* | "`nil`" .

63. *function-call* = *identifier* [*arg-list*] .

64. *variable* = *identifier*
| (*variable* ("[" | "(.") *expression* {"," *expression* }("]" | ".)"))
| (*variable* ("^" | "@"))
| (*variable* "." *identifier*) .

65. *member-designator* = *expression* [".." *expression*] .

Debugger & Profiler

Contents

Pascal–2 V2.0/RT-11 Debugger Guide

Introduction

The **Pascal–2** Debugger helps uncover program errors that cannot be caught at compilation time. These errors are usually ones in which the syntax is correct but the algorithm itself is not (e.g., incorrect number of loops in statements, unintended but legal changes to variable values, and the like). The Debugger is oriented to the programmer's viewpoint, so that you need no knowledge of the underlying computer architecture. When called, the Debugger will take control of a program. The Debugger keeps track of constants, variables, 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 it as the program executes. You can also access the original source text of your program for immediate identification of context. Taken together, these features allow the trouble-shooting of a program until you have detected and corrected any errors.

What the Compiler's Doing

The /debug compilation switch calls the Debugger. (See the Programmer's Guide for details on compilation switches.) The /debug switch causes several things to happen. The object module will contain extra code to locate statements in your program. The overhead is about one word per Pascal statement and about six words for each procedure.

The /debug switch also turns off optimizations that would interfere with debugging. For example, the compiler normally folds similar statements into one section of code and optimizes the usage of some variables by keeping their values in registers or on the stack temporarily. These optimizations would prevent 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.

The net effect of the Debugger overhead and the turning off of some optimizations is that programs will increase in size while you debug them. (See "Overlays" regarding what to do if the program size grows too much.) The code will return to its normal size once you correct the problem and recompile without /debug.

The /debug switch implies the /list switch, so the compiler will automatically generate a formatted listing file, .LST, 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 files with the extensions .SYM and .SMP, also in the same directory as the output file. The .SYM symbol table file describes the constants, types, and variables and the memory layout of variables. The symbol table file also contains information about each procedure and function. The .SMP file contains a map of the location of the statements and their position in the listing. Both the .SYM and the .SMP 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 link it with the Debugger. The Debugger is supplied in the PASCAL.OBJ library on the system device and will be automatically included when the /debug switch is used. See the examples below for the proper command sequence.

Running the Debugger

Basic Debugger Commands

When in control of a program, the Debugger prompts with the right brace "}" symbol. (This may print on upper-case-only terminals as the right bracket "]" character.) The Debugger accepts any of the single-character commands defined in the following sections, with parameters in parentheses. Numeric parameters are indicated by 'n', as in the command `S(n)`. You may have more than one command specified on a single line; commands must be separated by semicolons. The `?` (question mark) command will print a summary of Debugger commands. To exit from the Debugger, give the `Q` (quit) command, or type a Control-Z (`^Z`), or type the Control-C (`^C`) twice in a row. A single Control-C (`^C`) typed during program execution will stop the Debugger, thus permitting you to break into "infinite loops" in your program.

Stepping Through a Debugger Session

You will seldom use only a single Debugger command at any one session, so no single example can demonstrate the context in which certain commands are used or all of the ways in which certain commands relate. Our approach, therefore, is to begin by stepping through a sample program to demonstrate some of the common commands in a problem/example context. The sections that follow will describe in detail each Debugger command, including those not demonstrated in the sample program. Examples in later sections also provide context by showing several commands used together.

The sample program, `ROTAT.PAS`, prints an array of seven integers. You are then asked for a starting and ending position in the array. The program is supposed to rotate that section of integers to the left, with the left digit replacing the right digit.

After correcting any syntax errors discovered in a normal compilation, you compile a program with `/debug` and then link it with the Debugger and the Pascal support library, as shown:

```
.R PASCAL
*ROTAT/DEBUG

.LINK ROTAT,SY:PASCAL
```

Remember that the `/debug` switch implies `/list` and that this listing file is the only one that should be used with the Debugger. This listing file, `ROTAT.LST`, 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. You should have a printout of the listing file as reference when you begin a debugging session, or you can use the `L` command to list parts of the program while you are debugging.

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
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];
   17    4       A[Last] := A[First];
   18    5       write('Rotated ',first: 1,' thru ',last: 1,'=');
   19          end;
   20
   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);
   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        end.

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

After compiling and linking the program, you can now run it. The Debugger will take control of the program and enter command mode.

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? ROTAT
} G                                                    begin execution
 1 2 3 4 5 6 7
Left,Right? 1,7

?Array subscript out of bounds
 Program counter: 1356

Program terminated.

Breakpoint at ROTATE,3  A[I] := A[I + 1];
} 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. The **for** loop in the **Rotate** procedure is looping too many times. We reduce the final value by 1 (`last` becomes `last - 1` in line 15) and recompile the program.

This time, 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 -- 27-Jan-81

Program name? ROTAT
} L(Rotate,1,5)                                        list 5 lines of procedure Rotate
   14    1      begin
   15    2       for I := First to Last - 1 do
   16    3         A[I] := A[I + 1];
   17    4       A[Last] := A[First];
   18    5       write('Rotated ',first: 1,' thru ',last: 1,'=');
} G                                                    begin execution
 1 2 3 4 5 6 7
Left,Right? 1,7
Rotated 1 thru 7= 2 3 4 5 6 7 2
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
} W(N[7])                                              write the value of N[7]
2
```

The results are closer to what they should be, but something is still wrong: the last number is not 1 as it should be. 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[7])  ——————————————————————————————— write value of N[7]
7
```

(Note the way in which the Debugger counts statements when more than one is placed on a line, as on line 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[7] is changed.

```
} V(N[7])  ——————————————————— watch for changes of value of N[7]
} C  ———————————————————————————————— continue execution
Left,Right? 1,7
The value of "N[7]" was changed by the statement:
Rotate,4  A[Last] := A[First];
Old value: 7
New value: 2
Breakpoint at ROTATE,5 write('Rotated ',first:1,' thru ',last:1,'=');
} W(First,Last)  ——————————————— write values of First and Last
1  7
} W(N)  ————————————————————————————————— write values of array N
 2 3 4 5 6 7 2

}
```

At **Rotate,4** the first element is assigned to the last element after the first element has been changed. We must introduce a temporary variable to hold the first element value so that it will not be destroyed. We correct the program (adding a "temp" variable, an assignment at line 14 and another between lines 16 and 17), then recompile.

Again, we use the L command to inspect the part of the program we changed.

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? ROTAT
} L(Rotate,1,6)  ------------------------- list 6 lines of procedure Rotate
   14    1      begin Temp := N[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,'=');
   19          end;
} G  ------------------------------------------------------ begin execution
 1 2 3 4 5 6 7
Left,Right? 1,7
Rotated 1 thru 7= 2 3 4 5 6 7 1
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
} G  ------------------------------------------------------ begin execution
 1 2 3 4 5 6 7
Left,Right? 3,4
Rotated 3 thru 4= 1 2 4 3 5 6 7
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
} G  ------------------------------------------------------ begin execution
 1 2 3 4 5 6 7
Left,Right? 2,6
Rotated 2 thru 6= 1 3 4 5 6 2 7
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
} Q  ------------------------------------------------------ quit Debugger
```

Now the program seems to be running correctly. Note that the G command will restart the program even after it has terminated.

Once satisfied that the program is correct, we recompile it without the /debug switch to reduce memory requirements and to improve execution speed.

Debugger commands are described in detail in the next sections. Commands with similar functions are grouped together. Some examples are grouped together to provide context. Appendix A contains a summary of Debugger commands.

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. Statements are sequentially numbered within each block. Statement numbers are listed in the second column of the program listing produced by the /debug command.

The B(BlockName,StatementNumber) command sets a control breakpoint within the block named **BlockName** at the statement numbered **StatementNumber**. When the breakpoint is reached, your program will be interrupted before execution of the named statement, the breakpoint will be identified, and the Pascal source line will be displayed. The Debugger then will accept commands.

These may be interactive commands (from your terminal) or stored commands executed automatically. To save stored commands for automatic execution at a breakpoint, append a list of commands separated by semicolons and enclosed in angle brackets B()< ... >. Any Debugger command can 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 will continue without waiting for an interactive command.

You may interrupt the program at any time with a Control-C (^C). This command will stop the program and identify 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) will not take effect until after you have completed the input request.

A run-time error or program termination also will cause a control breakpoint after the error message or termination status is displayed. You may set any number of control breakpoints. (The program will execute more slowly if you define many.)

See the example listed after the C command.

K, K(): Killing of Breakpoints

You may remove a breakpoint in two ways. The B command with no parameters will delete the breakpoint that most recently stopped the program. Otherwise, the K(BlockName, StatementNumber) command will delete the breakpoint specified. The K command with no parameters will remove all breakpoints.

See the example after the C command.

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 will monitor a variable of any type, but only the first 32 bytes of data will be watched. You may watch any number of variables. (The program will execute slowly if you set many.)

Example of data breakpoints:

```
} V(N[7])
} C
Left,Right? 1,7
The value of "N[7]" was changed by the statement:
Rotate,4  A[Last] := A[First];
Old value: 7
New value: 2
Breakpoint at ROTATE,5 write('Rotated ',first:1,' thru ',last:1,'=');
```

(This example, ROTAT.PAS, is from "Running the Debugger", before.)

The **V** command without parameters removes all data breakpoints. It is not possible to remove individual data breakpoints.

If a local variable is being monitored and the associated block is completed, the Debugger will remove the breakpoint and display a message that the variable no longer exists.

Example:

```
Breakpoint at DECODE,3  FirstChar := M[1];
} L
 1255      function Decode(M:Name):integer;
 1256
 1257      var I: BoardIndex; Found: boolean; FirstChar: char;
 1258
 1259    1 begin
 1260    2   Found := false;
 1261    3   FirstChar := M[1];
 1262    4   if (FirstChar >= 'a') and (FirstChar <= 'h') then
 1263    5     FirstChar := chr(ord(FirstChar)-40);
 1264    6   for I := 5 to 40 do
 1265    7     if FirstChar = UserMoves[I,1] then
 1266    8       if M[2] = UserMoves[I,2] then begin
 1267    9         Found := true;
 1268   10         Decode := I;
 1269              end;
 1270   11   if not Found then Decode := 0;
 1271      end;
 1272
} V(Found)
} C
The value of "FOUND" was changed by the statement:
DECODE,9  Found := true;
Old value: FALSE
New value:  TRUE
Breakpoint at DECODE,10  Decode := I;
} K ------------------------------------------------ kill breakpoints
} C ------------------------------------------------ continue execution
Watch terminated for "FOUND".  Value did not change.
```

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. The G command may be used at any point in the program to restart it.

See the example after the C command.

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 can set a breakpoint at COUNT,10 inside a loop structure. When the Debugger stops at that breakpoint, you can 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 will function 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 will receive an error message telling you to use the G command to restart the program.

Examples for the B, K, D, G and C Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? 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,4)<W('In rotate, I=',i)>
} G
 1 2 3 4 5 6 7
Left,Right? 1,7
Breakpoint at MAIN,9  Rotate(Left, Right, N);
I=  4
Breakpoint at ROTATE,4  A[I] := A[I + 1];
In rotate, I=  1
} D                    ---------------------------------------------- display breakpoints

Breakpoints

ROTATE,4  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)
1
Breakpoint at ROTATE,4  A[I] := A[I + 1];
In rotate, I=  3
3
} K(Rotate,4)  ------------------------------ kill specified breakpoint
} C  ---------------------------------------- continue execution
Rotated 1 thru 7= 2 3 4 5 6 7 1
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
} 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 Debugger
```

S, S(): Step to Next Statement

The S (Step) command executes the next statement of the program. The S(n) command will execute 'n' statements without interruption. If a statement being "stepped" calls another procedure or function, that new procedure or function also will be 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 will not single-step through functions and procedures called by the current statement. P will treat an entire nested call as a single statement; thus procedure calls and function invocations will be 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 will begin 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 will receive an error message telling you to use G to restart the program.

Examples for the S and P Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? ROTAT
} B(main,9)
} G
 1 2 3 4 5 6 7
Left,Right? 1,5
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} S
Breakpoint at ROTATE,1  begin Temp := N[First];
} S
Breakpoint at ROTATE,2  begin Temp := N[First];
} S
Breakpoint at ROTATE,3  for I := First to Last - 1 do
} S
Breakpoint at ROTATE,4  A[I] := A[I + 1];
} S
Breakpoint at ROTATE,4  A[I] := A[I + 1];
} S(3)
Breakpoint at ROTATE,5  A[Last] := Temp;
} C
Rotated 1 thru 5= 2 3 4 5 1 6 7
Program terminated.
Breakpoint at MAIN,11  write(N[I]:2);
} G
 1 2 3 4 5 6 7
Left,Right? 1,5
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} P
Rotated 1 thru 5=Breakpoint at MAIN,10 for I := 1 to Arraylen do
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P
 2Breakpoint at MAIN,11  write(N[I]:2);
```

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 can review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the `H` command will show 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)` will print 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 will be identified by its block name and statement number and will be displayed before being executed.

A Control-C (^C) will interrupt the trace and return the Debugger to command mode. You can then turn off tracing with the T(FALSE) command and continue running your program with the C command.

Example for the T Command

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? 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)<T(TRUE);C>  ---------- set breakpoint, tracing, continue
} G
 1 2 3 4 5 6 7
Left,Right? 1,3
Breakpoint at MAIN,9  Rotate(Left, Right, N);
   ------------------------------------------------------ tracing enabled
ROTATE,1  begin Temp := N[First];
ROTATE,2  begin Temp := N[First];
ROTATE,3  for I := First to Last - 1 do
ROTATE,4  A[I] := A[I + 1];
ROTATE,4  A[I] := A[I + 1];
ROTATE,5  A[Last] := Temp;
ROTATE,6  write('Rotated ',first: 1,' thru ',last: 1,'=');
Rotated 1 thru 3=MAIN,10  for I := 1 to Arraylen do
MAIN,11  write(N[I]:2);
 2MAIN,11  write(N[I]:2);
 3MAIN,11  write(N[I]:2);
 1MAIN,11  write(N[I]:2);
 4MAIN,11  write(N[I]:2);
 5MAIN,11  write(N[I]:2);
 6MAIN,11  write(N[I]:2);
 7
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
```

```
} T(FALSE)                                                    tracing off
} D

Breakpoints

MAIN,9  Rotate(Left, Right, N);
      <T(TRUE);C>

} K
} G
 1 2 3 4 5 6 7
Left,Right? 1,3
Rotated 1 thru 3= 2 3 1 4 5 6 7
Program terminated.

Breakpoint at MAIN,11  write(N[I]:2);
```

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 to write. As shown, you can 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 16-bit signed decimal numbers. Set variables are displayed in Pascal set notation. Scalar variables are displayed as the name of the enumerated type 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 an octal 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 can be 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 16-bit signed integers.

The Debugger can write any complex Pascal data structure, including records and arrays. Each element of an array is printed, starting with the first element.

Each change of a multi-dimensional array index generates an end-of-line character, so that the array or table is clearly displayed in groupings that correspond to changes in the indices. One blank line will separate the elements of the rightmost index from those in the next-rightmost index; two blank

lines will separate the elements of the next-rightmost index from those in the third-rightmost index; and so on.

Example:

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
MULTI/DEBUG

 Line Stmt
    1      program Multi;                    { multi-dimension 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 -- 27-Jan-81

Program name? 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 will list 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 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.

Examples for the W Command and Variable Assignment

```
Pascal-2 RT-11 SJ V2.0H    5-Apr-81   7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
 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 -- 27-Jan-81
Program name? COLOR
} G

Program terminated.

Breakpoint at MAIN,7 R.I := 123; R.S := [Orange, Green]; R.C := 'TEST';
} 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 breakpoint locations with stored commands, user-defined macros, and the variables being watched. 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.PAS 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 via 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 can list any other procedure by giving the procedure name enclosed in parentheses. For example, L(main) will list the body of the main program.

The command L(Proc, Stmt) lists a single statement, where 'Proc' is the name of the procedure and 'Stmt' 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) will list the first ten lines of the main program.

The general form of the command is L(Proc, Stmt, Count) where 'Proc' and 'Stmt' describe a statement in the program. A positive 'Count' will print that many lines starting at the statement specified. A negative 'Count' will print that many lines in front of the statement specified.

This example lists 5 lines beginning with the 1st statement of procedure **Rotate**:

```
} L(Rotate,1,5)
   14    1      begin
   15    2        for I := First to Last - 1 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 before the start of the 4th statement of procedure **Rotate**:

```
} L(Rotate,4,-2)
   15    3        for I := First to Last - 1 do
   16    4          A[I] := A[I + 1];
```

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 will not always print the complete header information and sometimes may print part of the preceding procedure.

Long procedures may take some time to print. A single Control-C (^C) will interrupt the listing and return the Debugger to command mode.

Utility Commands

M(): Define Macro

The **M** command can save 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 symbol containing up to 32 letters. The **X** command uses 'Name' to identify the macro. You may place as many Debugger commands in the angle brackets "< ... >" as will fit on one command line. You can 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 will list 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.

Examples for the M and X Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 27-Jan-81

Program name? ROTAT
} M(DumpN)<W('The value of N=',N)>          ——————————— define macro
} B(Rotate,1); G
 1 2 3 4 5 6 7
Left,Right? 1,7
Breakpoint at ROTATE,1  begin Temp := N[First];
} M(DumpI)<W('I=',I)>          ——————————————————————— define macro
} D
Breakpoints

ROTATE,1  begin Temp := N[First];

Macros

DUMPI       W('I=',I)
DUMPN       W('The value of N=',N)
} S
Breakpoint at ROTATE,2  begin Temp := N[First];
} S
Breakpoint at ROTATE,3  for I := First to Last - 1 do
} S
Breakpoint at ROTATE,4  A[I] := A[I + 1];
} X(DumpI)          ——————————————————————————— execute macro
I=  1
} S(4); X(DumpI); X(DumpN)
Breakpoint at ROTATE,4  A[I] := A[I + 1];
I=  5
The value of N=  2 3 4 5 5 6 7

} Q
```

Execution Stack Commands

The execution stack command **H** shows at any time a history of program execution and the current stack of pending procedure and function calls; the **N** command lists the names of the parameters and local variables in any procedure in the execution stack; and the **E** command allows you to change the context of the stack frame from the current procedure to another so you can access variables you otherwise wouldn't be able to.

H, H(): History of Program Execution

As described above, the Debugger maintains a list of the last 50 statements executed while your program was running. With the **H** command you can review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the **H** command will show 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)** will print 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) will print 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 N 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 (see below), this 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 "*" character are those procedures which lexically contain the definition of the current procedure. The parameters and local variables in the procedures so marked 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 will display 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 will list 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 value of variables unless they are in procedures or functions marked with the "*".

The E command, described below, 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's scope rules. If you stop your program in the middle of a procedure, you can write or modify only the variables and parameters in each of the procedures enclosing the current procedure, as described above in the section on the H command.

If you want to look at or change local variables in procedures that are not accessible to the current procedure, the E command will get 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 a "*" (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 will make procedure 5 the current procedure. Then, using the context of procedure 5, the Debugger will print the value of the variable 'I'. At the end of the command line, the current procedure will be changed back to the top procedure in the execution stack.

Because the **N** command allows you to list the names (only) from all the procedures on the execution stack, the following commands are equivalent:

```
} E(5); N
} N(5)
```

Examples for 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);

}
```

Overlays

Debugger modules, contained within the **Pascal–2** library, are automatically included during the linking of a module compiled with the /DEBUG switch. These modules can add up to 12K words to the size of a program, often making it too big to run. If this happens (as it often will with large programs), overlay the Debugger so that it takes less room, as shown:

```
.LINK/PROMPT
*PROG = SY:DBRUN,SY:PASCAL/C/T
*PROG,SY:DBUSER/O:1/C
*SY:DBUG/O:1
Transfer address? $START
```

DBRUN.OBJ, the interface between **Pascal–2** programs and the **Pascal–2** Debugger, processes all Debugger traps. DBRUN.OBJ must be linked into the root segment of overlaid programs.

DBUSER.OBJ reloads the user program after an exit from Debugger command mode. DBUSER.OBJ must be loaded into the same region as the program being debugged.

DBUG.OBJ is the Debugger, which processes Debugger commands. The Debugger normally overlays the user program being debugged.

Appendix A: Debugger Command Summary

B	Remove current breakpoint
B(Block,Statement)	Set a control breakpoint
B(Block,Statement)< ... >	Control breakpoint with stored commands
C	Continue through breakpoint
C(n)	Continue 'n' times
D	Display breakpoints and macros
E(n)	Enter context of frame 'n' (1 line only)
G	Restart program
H	Display recent history and full stack
H(n)	Display last 'n' statements
K	Remove all control breakpoints
K(Block, Statement)	Remove specified breakpoint
L(Proc)	List source of **Proc**
L(Proc, n)	List statement 'n' in **Proc**
L(Proc, n, x)	List 'x' lines after statement 'n' in **Proc**
M(Name)<Commands>	Define stored command macro
N	List variables for current frame
N(n)	List variables 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

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 and prints a summary describing how many times each procedure is called, and what fraction 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's Guide for details on compilation switches.) The `/profile` switch will cause 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 and are described in "What the Compiler's Doing" at the beginning of the Debugger Guide. The Profiler requires 2K words, plus about 4 words per statement in the program. Thus, a program containing 1000 statements will require about 6K words for the Profiler.

After compiling the program, include the Profiler module at the link step. The Profiler is supplied as an object module, `SY:PRFILE.OBJ`. The compilation and link steps are shown below, using a program named `PROG.PAS`.

```
.R PASCAL
*PROG/PROFILE

.LINK PROG,SY:PRFILE,SY:PASCAL
```

The Profiler will take control of the program and ask for its name. The name should be the name of the output file specified when you compiled the program. This name is used to open the 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 will ask for the name of the profile output file. If you specify a disk file, the default file type will be `.PRO`. Writing a profile to the terminal is practical only for a short program.

The next example involves a program, `CHECKR.PAS`, that plays a game of checkers. Compile and link the program as described above, then run it.

```
.RUN CHECKR
profile  V3.1  15-Mar-81

Program name? 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 will slow down program execution. For this reason, it may not always be possible to profile programs that operate in a time-critical environment.

The Profiler will generate 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) will interrupt the program and generate a profile at that point. Two control-C's will abort 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, will appear 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 will 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 example profile below 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 will be documented 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 example above because the CHECKR program uses random numbers to play a different game each time it is run.) Overall, the Program Execution Summary will show 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.

Utilities

Contents

Pascal–2 V2.0/RT-11 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. 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 on its use. Briefly, however, 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; and PROCREF, which cross-references the procedures in your program.

Dynamic String Package: 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 will accept standard Pascal and the language extensions in **Pascal–2**. PASMAT accepts full programs, external procedures, or groups of statements. A syntactically incorrect program will cause 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 formatter directives give you considerable control over the output format when you wish.

Overview of Capabilities

PASMAT has these capabilities:

1. The program may be converted to uniform case conventions, under the control of the user.
2. The program is indented to show its logical structure and to fit into a specified output line length.
3. Comment delimiters are changed to braces ({}).
4. If requested, the break character (_) will be removed from identifiers for use at installations that do not support the break character.
5. If requested, the first instance of each identifier will determine the appearance of all subsequent instances of the identifier.
6. All nonprinting 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.

1. A comment standing alone on a line will be left-justified to the current indention level, so that it will be aligned with the statements before and after it. If it is too long to fit with this alignment, it will be right-justified.
2. A comment that begins a line and continues to another line will be 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.
3. If a comment covered by one of the above rules will not fit within the defined output line length, the output line will be extended as necessary to accommodate the comment. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence.
4. A comment embedded within a line will be 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 will fit within the output length, that line will be extended as necessary. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence. If no code follows a comment in the input line, then no code will be 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 below) tells the formatter to place these statements on the same line as the **case** labels if the statements will 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 will fit.

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 will be 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:

```
.R PASMAT
*output-file = input-file /options="directives"
```

input-file: 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: The reformatted output file. If *output-file* is omitted, the output file receives the same name as the input file and becomes the latest version of that file.

/options="*directives***"**: Settings for formatting directives. The /options switch is optional. It may be abbreviated to /o and may be placed anywhere on the command line. When specified on the command line, directives must be placed in quotes as shown. The *directives* field will be scanned as though the directives were in a Pascal comment at the start of the source program.

Formatting Directives

Formatting directives can 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 breeds: 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.

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.

```
.R 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 break character (_) in program text.

`B` (Default `B-`) Specifies that the statements following a **`then`**, or **`else`**, **`for`**, **`with`** or **`while`** will be put on the same line if they will fit. The statement following a **`case`** label will be put 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 the Pascal-1 **`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 will be 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 will not fit in the width specified, the line will be lengthened accordingly, and a message at the end of the formatting will give the number of times the width was exceeded and the output line number of the first occurrence.

`P` (Default `P-`) Sets "portability mode" formatting, which removes break characters (_) from identifiers. The first letter of each identifier, and the first letter following each break character, will be made upper case, while the remaining characters will be in lower case. This directive overrides the `L` and `U` directives. The `R` directive sets the case of reserved words.

Warning: **Pascal–2** considers break 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 will be in upper case. With this off, reserved words will be in lower case. The `L` directive overrides the `R` directive.

`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 will not be 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 will be additionally indented by half the value of `T`.

`U` (Default `U-`) `U+` specifies that identifiers are converted to upper case; `U-` specifies that they will be converted to lower case. The `L` and `P` directives override this directive.

Limitations and Errors

PASMAT is limited in these ways:

1. The maximum input line length is 132 characters.
2. The maximum output length is 132 characters.
3. Only syntactically correct statements are formatted. A syntax error in the code will cause the formatting to abort. An error message will give 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 will compile.
4. The number of indention levels handled by PASMAT is limited; PASMAT will abort if this number is exceeded — a rare circumstance.
5. If a comment will require more than the maximum output length (132) to meet the rules given, processing will be aborted. This situation should be even rarer than indention-level problems.
6. 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 will recover all the text for every error condition.

PASMAT Examples

To show how the various PASMAT options work together, we will take the sample program that follows and show how it appears after reformatting with two different sets of options.

```
program Ex(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 /options switch on the command line to specify the width of the output line (the options switch specifies a short line width to illustrate the right-justification of long comments).

The program is formatted with the commands:

```
.R PASMAT
*EX/O:"O=66"
```

Program text after formatting:

```
program Ex(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:

```
.R PASMAT
*EX
```

Program text after formatting:

```
{[A+,L-,R+]}
PROGRAM Ex(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.
```

PB: A Pascal–2 Formatter

PB is designed on the premise that a formatting program can't do everything, that formatting requires an understanding of the meaning of a program. Thus, PB is meant to assist, rather than replace, the manual arrangement of program format. The simple transformations performed by PB reduce the tediousness of formatting program text, and help ensure consistency within a variety of personal formatting styles.

PB can be used on code as it is being developed, even code that is incomplete or incorrect. You write a program, or program fragment, to some level of detail, then run it through PB. You can then edit the resulting code to alter its meaning or improve its appearance, and use PB again. This cycle can be repeated as the code progresses from initial idea to working program, and later as the program is "maintained".

Text produced by PB usually looks much like the input. Each input line is transformed into a single output line containing essentially the same text; within the code on a line the spacing is the same; and simple statements that continue onto multiple lines stay lined up.

PB adjusts program format to be consistent with the syntactic structure of Pascal. Statements at the same nesting level line up, and indention increases with the nesting level. Where possible, trailing comments are lined up with one another. Keywords and identifiers in the text are altered to match the capitalization style of their first occurrence (which may be in an included file).

Using PB

You invoke PB with the following command:

```
.R PB
* output-file = input-files
```

input-files: These are the Pascal source files being reformatted (the default file extension is .PAS). Multiple files, if specified, are separated by commas. Multiple files are concatenated to produce the output file.

output-file: file in which the reformatted Pascal source file is placed (default file extension is .PAS). If the *output-file* is not specified, it will be created with the same name as the last input file.

Two command line switches adjust PB formatting. The indent switch (/indent:*num*) specifies the number of columns (*num*) in an indention step (the space that text is shifted when the nesting level changes). The default setting is 2; larger values make the separation between levels clearer, but may force text past the right margin. The comment indent (/comment:*num*) specifies the column (*num*) to which trailing comments are indented. The default setting is 33; this value works well when trailing comments are used primarily to annotate declarations.

These command line switches are placed after the input file names.

Example

This example illustrates the functions provided by PB. The example also shows a particular development style, discussed above, to which PB is suited.

We start with the program at an intermediate stage in its development. It is perhaps one or two stages past an initial sketch, and some new code has just been added. Notice that the new code is not indented; it is broken into reasonable lines, but we will let PB do the rest of the formatting.

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
repeat
x := x + 1;
prim := x is a prime number;
until prim;
write(X);
  end;
end.
```

Processing by PB gives this result:

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
    repeat
      X := X + 1;
      prim := X is a prime number;
    until prim;
    write(X);
    end;
end.
```

The indent changes at most one step from line to line. When control constructs appear one per line (the **repeat** statement, for instance) each causes the indent to increase by one step, but when a second one appears on a line it has no effect on indention. In the example, this has been used to cut the "noise" from **begin**...**end** brackets.

Notice a couple of things at this stage. First, PB does not know much about Pascal language syntax (note the phrase "`X is a prime number`"). Second, PB uses the first instance of a word, regardless of context, for its capitalization style (you can set the style for an identifier by adjusting its declaration, since the declaration of an identifier must precede its use).

Of course, the program is not yet complete, nor does it contain any comments. The following is closer to a final version.

```
type Index = 1..n;
var
X: integer; { number being tested for primality }
j, { count of primes found }
k, { trial divisor }
lim: index; { last divisor to test }
Prim: boolean; { 'true' until a divisor is found }
P: array[index] of integer; { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; Lim := 1;
write('2');
  for J := 2 to n do begin
    repeat
      X := X + 2;
if sqr(P[Lim]) <= X then Lim := Lim + 1;
K := 2; Prim := true;
while Prim and (K < Lim) do begin
Prim := (X mod P[K]) <> 0;
K := K + 1;
end;
    until Prim;
    P[J] := X; write(X);
    end;
end.
```

Processing by PB gives:

```
type Index = 1..n;
var
  X: integer;                    { number being tested for primality }
  j,                             { count of primes found }
  k,                             { trial divisor }
  lim: Index;                    { last divisor to test }
  Prim: boolean;                 { 'true' until a divisor is found }
  P: array[Index] of integer;    { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; lim := 1;
  write('2');
  for j := 2 to n do begin
    repeat
      X := X + 2;
      if sqr(P[lim]) <= X then lim := lim + 1;
      k := 2; Prim := true;
      while Prim and (k < lim) do begin
        Prim := (X mod P[k]) <> 0;
        k := k + 1;
        end;
    until Prim;
    P[j] := X; write(X);
    end;
end.
```

The comments have been moved to the right, where they stand apart from the program code and line up for easier reading. If we had wanted to keep a comment attached to the code, we could have placed it in front of the final comma or semicolon on its line (then it would not be a trailing comment and would be treated as part of the text), or placed it on a line by itself (where it would be aligned at the prevailing indent).

Detailed Formatting Rules

Indention is directed by the nesting of control constructs in the program text. Generally, when the nesting level increases, the indent increases by the indention step; when a nesting level ends, the indent reverts to that of the surrounding nesting level. A change of indention at the beginning of a line takes effect immediately; otherwise, it takes effect on the next line. The exceptions to the rules are:

1. The start of a new nesting level does not change the indent if it begins on the same line as the surrounding level.
2. When a line begins with a statement label, the indent for that line is decreased by the indention step.

Normal indention rules do not apply when a simple statement or clause continues across multiple lines. In these cases, the initial line is indented normally, but the following lines are arranged to preserve their alignment with the initial line. Changes of indention within continued lines take effect after the last continuation.

A similar adjustment occurs when a comment continues across multiple lines. When a trailing comment is continued, the following lines stay aligned with the initial part of the comment, but not necessarily with the rest of that line (a trailing comment may shift in relation to the rest of the line).

The following constructs affect the indention level:

1. A *program-declaration*, *procedure-declaration* or *function-declaration* is arranged so that the *heading*, the keyword introducing a *label-declaration-part*, *const-definition-part*, *type-definition-part*, or *variable-declaration-part*, and the *body* are all at the same indention level. The list of declarations within a *label-declaration-part*, *const-definition-part*, *type-definition-part*, *variable-declaration-part*, or *procedure-and-function-declaration-part* is set one indention step deeper.
2. The *component-type* of an *array-type* or *file-type*, and the *base-type* of a *set-type* are indented one more step. Within a *record-type* the *field-list* is indented another step, the list of *variants* within the *variant-part* is indented an additional step, and the *field-lists* within individual *variants* are indented yet another step.
3. The *statement-sequence* within a *compound-statement* is indented an additional step.
4. The controlled *statement* within a *for-statement*, *if-statement*, *else-part*, *while-statement* or *with-statement*, and the *statement-sequence* within a *repeat-statement* are indented another step. Within a *case-statement* the list of *case-list-elements* is indented one more step, and the controlled *statement* of each *case-list-element* is indented an additional step.

XREF: A Pascal–2 Cross-Reference Lister

XREF produces a cross-reference listing of the identifiers in a Pascal program. 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:

```
.R XREF
*output-file = input-file /switch
```

input-file: The Pascal source file being cross-referenced. The input file has a default extension of .PAS. XREF accepts only one input file.

output-file: The cross-reference file. The output file has a default extension of .CRF. *Output-file* and = are optional. If they are omitted, an output file with the same name as the input file is placed in the default directory.

/switch: Command line switches. *Switch* can be either or both of these 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 simplifies the locating of 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 characters.

The switches may be abbreviated to one letter.

Limitations

The XREF program has two limitations on the size of the programs it can handle.

1. An internal limit exists for the number of distinct identifiers allowed. You can change this number in XREF.PAS and recompile the program.
2. The total number of references is limited by the amount of dynamic storage available.

The XREF program does not do a complete syntax analysis of the program, and it may not flag all declarations or assignments.

Example of XREF Listing

This example shows the cross-referencing of the program EX.PAS.

```
.R XREF
*EX/LIST/WIDTH:66
```

```
      1 program Ex(output);
      2 { Compute an approximation for E from its Taylor series
c     3   The Nth term in the series is 1/(N!)
c     4 } ———————————— c is the flag character for continued comments
      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     { loop to approximate E; quit when the sum stops changing }
     16     repeat
     17       E := E + SeriesTerm;
     18       { compute next term of series }
     19       N := N + 1;
     20       SeriesTerm := SeriesTerm / N;
     21     until E = (E + SeriesTerm);
     22     writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
     23   end.

Cross reference:  * indicates definition, = indicates assignment

-E-
E               7*    12=    17=    17     21     21     22
EX              1*

-I-
INTEGER         8

-N-
N               8*    13=    19=    19     20     22

-O-
OUTPUT          1*

-R-
REAL            7

-S-
SERIESTERM      7*    14=    17     20=    20     21

-W-
WRITELN        22


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 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:

1. The file and line number where its heading starts.
2. 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.
3. If the procedure was declared **forward** or is externally defined, the listing contains the file and line number where the procedure heading stub starts.
4. 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.
5. 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.

Use of PROCREF

You invoke PROCREF with the following command:

```
.R PROCREF
* output-file = input-files /width:num
```

input-files: 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 will be concatenated.

output-file: The cross-reference file. The output file has a default extension of .PRF. *Output-file* and = are optional. If they are omitted, an output file with the same name as the last input file 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 it may be abbreviated to one letter.

Note that the RT-11 system will automatically truncate the name PROCREF to PROCRE.

Limitations

The PROCREF program does not do a complete syntax analysis of the program being processed. PROCREF will err 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 will be treated as a reference to the procedure.

Example

Let's assume that we wish to generate a procedure cross-reference for the following program, LVSPOOL.PAS.

```
Pascal-2 RT-11 SJ V2.0H  19-Apr-81   6:21 AM    Site #1-1    Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
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.

```
.R PROCREF
*LVSPOOL = LVSPOOL/W:72
```

The /W:72 requests that the cross-reference listing not exceed 72 characters wide so that we can look at the result on a terminal. The result, placed in the file LVSPOOL.PRF, consists of:

```
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

A package of procedures and functions for dynamic string processing, STRING.PAS, is supplied in the **Pascal–2** distribution kit. The package, written in standard Pascal, provides convenient facilities for handling character strings of varying lengths. Also, programs using STRING may easily be moved to other Pascal implementations.

Strings are stored as a record structure with space for a fixed maximum number of characters (normally 100 but easily changed), and an integer marking the current length of the string:

```
type String = record
  Len: integer;
  Ch:  packed array[1..StringMax] of char;
  end;
```

In the definitions below, *string* and *target* represent variables of type String. *File* is a variable of type text. *Start* and *span* are integers.

The capabilities provided are:

Len(*string*): Returns the length of *string*.

Clear(*string*): Initializes *string* to empty.

ReadString(*file*, *string*): Reads *string* from *file*. The string is terminated by eoln(*file*), and a readln(*file*) is performed. A string longer than the maximum allowed, StringMax, is truncated.

WriteString(*file*, *string*): Writes *string* to *file*. The same effect can be achieved by passing the parameter *string*.Ch: *string*.Len to write, as in write(F, 'S=', S.Ch: S.Len).

Concatenate(*target*, *string*): Appends *string* to *target*. The resulting value is *target*. Overflow results in truncation to StringMax characters.

Search(*string*, *target*, *start*): Searches *target* for the first occurrence of *string* to the right of position *start* (characters are numbered beginning with 1). The function Search returns the position of the first character in the matching substring, or the value zero if *string* does not appear.

Insert(*target*, *string*, *start*): Inserts *string* into *target* at position *start*. Characters are shifted to the right as necessary. Overflow produces a truncated target string. A *start* position that would produce a non-contiguous string has no effect.

The *start* and *span* parameters in the Substring and Delete 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.

Delete(*string*, *start*, *span*): Deletes the substring defined by *start*, *span* from *string*.

Substring(*target*, *string*, *start*, *span*): The substring of *string* defined by *start*, *span* is assigned to *target*.

Example

The following sample program, EXAMPL.PAS, demonstrates the use of the string package to read a line from the terminal and to separate the string into single words. The string package may be used as a "header" file or included in the program text via the **%include** command. If the string package is used as a header file, the command is:

```
.R PASCAL
*STRING,EXAMPL
```

The following example uses STRING.PAS as an included file.

```
program Exampl;
%include String;  ------------------------------ dynamic string package

type Lit = packed array [1..10] of char;
var  Space, Line, Word: String;
     Mark: integer;

  procedure Literal(var T: String; Ch: Lit; N: integer);
  var I: integer;
  begin
    Clear(T);  ------------------------------------ defined in string.pas
    for I := 1 to N do T.Ch[I] := Ch[I];
    T.Len := N;
  end;

begin
  Literal(Space,'          ', 1);  ---------------- make 1 char string
  write('Type a line: ');
  ReadString(Input, Line);  ---------------------- defined in string.pas
  Concatenate(Line, Space);  --------------------- defined in string.pas
  Mark := Search(Line, Space, 1);  --------------- defined in string.pas
  while Mark > 0 do begin
    Substring(Word, Line, 1, Mark - 1);  ---------- defined in string.pas
    if Len(Word) > 0 then begin
      WriteString(Output,Word);  ----------------- defined in string.pas
      writeln;
      end;
    Delete(Line, 1, Mark);  ------------------------ defined in string.pas
    Mark := Search(Line, Space, 1);
    end;
end.
```

Run the program with these commands:

```
.R PASCAL
*EXAMPL

.LINK EXAMPL,SY:PASCAL
.RUN EXAMPL
Type a line: This is an example.
This
is
an
example.
```

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:

1. 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.)
2. Isolate a common function and make the procedure handle the most general case of that function.
3. 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's Guide.

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:

Name	Arguments	Function
`proc`	*procname*	Begin the declaration for the procedure *procname*.
`func`	*funcname* *result* *restype*	Begin the declaration for the function *function*. The returned value will be that assigned to *result*, of type *restype*.
`param`	*parmname* *parmtype*	Declare a parameter named *parmname* of type *parmtype*.
`var`	*varname* *vartype*	Declare a local variable named *varname* of type *vartype*.
`save`	<*reg0*, . . . ,*regn*>	Specify general registers to save on procedure entry.

rsave	*<ac0, ... ,acn>*	Specify floating accumulators to save on procedure entry.
begin		Begin the actual procedure code. This macro generates code to push the variables on the stack and to save registers.
endpr		End the code for this procedure, restore registers, pop variables and parameters from the stack, and return to the calling location.

The following example demonstrates how these macros may be used in a procedure declaration. Note the correspondence between the **Pascal–2** code and the MACRO-11 code.

Pascal–2 declaration:

```
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}
```

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
```

Procedure Definition Macros

The PASMAC procedure definition macros must be used in the order:

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 will be produced 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 2, the identifier `real` has the value 4, and a disk buffer may have the value 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. Take into account any additional values pushed onto 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]
```

procname: Name to be used to call the procedure. Only the first six characters of this name are significant.

`check`: 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`]

funcname: Name to be used to call the function. Only the first six characters are significant.

resname: 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: 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`: 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*

paramname: 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: The length of the parameter, 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
```

varname: 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: The length of the variable, 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>
```

<reg1, ..., regn>: 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 will use 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]
```

<accum1, ..., accumn>: 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: 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

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.
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
```

Example

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 will begin with a version of the algorithm as it is coded in **Pascal–2**:

```
{$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 the unsigned number, which is a **Pascal–2** extension, avoids the shifting of the sign bit into the lower-order bits. 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:

1. It performs a single function with simple internal logic.
2. It is a generally useful form of the function, rather than a special use.
3. 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  examp

; 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 r0 = 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 `restore` macros would be used to save and restore the registers used. The following version is an example of this

kind of code.

```
        .title  examp1

        ; 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 works with local quantities 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
```

Use of PASMAC

The macros described here 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:

```
.R MACRO
*EXTPRO,EXTPRO=SY:PASMAC,EXTPRO
```

The result of this assembly is an object file (.OBJ) that is linked in the same way as any other external module. Note that the reference to any module assembled with the PASMAC package is **external** rather than **NonPascal**.

The example 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 P$... and macros of the form $P... for storing state data. Avoid such forms in your own code.

Contents

Prose: A Text Formatter

Introduction

The tediousness of preparing and editing computer-oriented documentation can be made easier by computerized text-processing tools such as text editors and formatters. This guide describes a text-formatting program named **Prose** and assumes a basic knowledge of computer-related text processing tools.

The **Pascal–2** distribution kit includes the **Pascal–2** manual in machine-readable form. The text has been prepared with **Prose**, and the **Prose** program is supplied in the **Pascal–2** system. **Prose** will allow you to print this (and any other) documentation in whatever format you wish.

The contents of this guide are adapted from the **Prose Instruction Manual** written by John P. Strait (© 1978, University Computer Center, University of Minnesota), as described in **Pascal News** (No. 15) in September 1979.

Text examples in this manual have been extracted from **Alice's Adventures In Wonderland** by Lewis Carroll.

Notes for the PDP-11

This document describes the version of **Prose** installed on the Digital PDP-11 and VAX 11/780 computers at Oregon Software. The default form of **Prose** commands correspond to PDP-11 usage, and the output devices fit those available on the PDP-11. Specific machine-dependent characteristics of this version of **Prose** are are flagged by "(**PDP-11**)". Several items in the original **Prose** manual referring to the idiosyncracies of another machine have been shamelessly removed.

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` went through many changes, stablizing somewhat in early 1977 at version 5.0, which is written in CDC COMPASS assembly language. **Prose**, written in Pascal, was developed 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. He converted this version from the Univac to the PDP-11 in July 1980 at Oregon Software.

Philosophy, Goals, and Capabilities

Prose is intended primarily for the preparation of machine-retrievable documentation, and this has influenced the choice of its repertoire of abilities. `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" [**Typeset 5.0 Information**, © 1977 by Michael Huck]. In spite of these aspirations, no program can be all things to all people,

and so it is with **Prose**. It is intended that **Prose** be able to do **most** of the things needed to produce high-quality computerized text.

The design of **Prose** had several goals, among them, that high-quality results should be produced with a minimum number of directives and that **Prose** should have about 90% of the abilities that you think are useful and that the 10% it doesn't have should be so esoteric as to be non-essential. Some text formatters take the approach of providing a minimum set of built-in abilities, along with a "general and powerful" feature such as macros. The idea is that you can accomplish anything you want (no matter how much effort it will take) by defining appropriate macros. The problem with this approach is that the user is forced to learn a complicated feature to produce any but the most trivial results.

The philosophy behind **Prose** is that the user should not be overwhelmed by a large number of complicated directives; that the syntax of the directives should be consistent; that the text, not the directives, should stand out. Because of this goal of simplicity, **Prose** may or may not be the tool for a given application. The following two tables should aid you in deciding whether to use **Prose**.

Prose . . .

a. **Prose** has a small number of commands providing an easily learned set of basic formatting abilities.

b. **Prose** can do underlining and discretionary hyphenation.

c. **Prose** can remember and restore the text processing environment.

d. **Prose** can produce mixed-case or upper-case-only output from either mixed-case or upper-case-only input.

e. **Prose** can accumulate and produce a sorted index, referring to page numbers.

f. **Prose** can print selected pages on request.

g. **Prose** can format text in pages with headers, footers, and other aids.

h. **Prose** can fill and justify text to specified margins.

i. **Prose** is a portable program, written in standard Pascal, using ASCII as its internal character code. **Prose** is written to encourage its transportation between computers with different hardware and different operating systems.

. . . and Cons

a. **Prose** cannot control photo-typesetting machines.

b. **Prose** cannot do graphics.

c. **Prose** does not have multi-column ability.

d. **Prose** does not have macros, variables, or other programming language-like features.

d. Except for its indexing ability, **Prose** cannot store text and retrieve it later.

e. **Prose** does not have tabs.

f. **Prose** does not have directives to do everything you always wanted to do.

Basic Units of Text

Some of the basic units of natural language are the word, the phrase, the sentence, and the paragraph. In text formatting, the basic units are the word, the line, and the paragraph. 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 it is next to. By default, **Prose** reformats its input by

filling words into **lines**, adding blanks to justify the lines to left and right margins, and printing lines together to make **paragraphs**. In filling lines, **Prose** does not pay any attention to the original positions of the words, but instead fills as many words as possible into the output lines, preserving the original order. The following example illustrates this process of filling and justifying.

Input to **Prose**:

```
    "When we were little," the Mock Turtle went on at last,
more calmly, though still sobbing a little now and then,
"We went to school in the sea.  The master was an old
Turtle--we used to call him Tortoise--"
    "Why did you call him Tortoise, if he wasn't one?"
Alice asked.
    "We called him Tortoise because he taught us," said the
Mock Turtle angrily.  "Really you are very dull!"
    "You ought to be ashamed of yourself for asking such a
simple question," added the Gryphon; and then
they both sat silent and looked at Alice, who felt ready to
sink into the earth.
```

Output from **Prose**:

```
    "When  we  were  little,"  the Mock Turtle went on at
last, more calmly, though still sobbing a little now  and
then,  "we  went to school in the sea.  The master was an
old Turtle--we used to call him Tortoise--"
    "Why did you call him Tortoise, if  he  wasn't  one?"
Alice asked.
    "We  called  him Tortoise because he taught us," said
the Mock Turtle angrily.  "Really you are very dull!"
    "You ought to be ashamed of yourself for asking  such
a simple question," added the Gryphon; and then they both
sat silent and looked at Alice, who felt  ready  to  sink
into the earth.
```

Most of text formatting is **filling** and **justifying**. In the absence of special instructions (called **directives**), **Prose** will fill all of the input words into output lines and justify all of those lines.

The distinction between one paragraph and the next is defined by a **justification break**, which causes **Prose** to stop filling the current output line and print it without justifying. Since the **break** is one of the most frequently used instructions (as well as one of the simplest), it can be indicated in many ways. Paragraphs can be separated (broken) by one or more blank lines, by leading blanks typed on an input line (a paragraph indentation), or by the **Prose** .BREAK directive. The following example demonstrates these three methods.

Input to **Prose**:

```
At last the Gryphon said to the Mock Turtle "Drive on,
old fellow!  Don't be all day about it!" and he went
on in these words:--

"Yes, we went to school in the sea, though you mayn't
believe it--"
.BREAK
"I never said I didn't!" interrupted Alice.
.BREAK
"You did." said the Mock Turtle.
    "Hold your tongue!" added the Gryphon, before Alice
could speak again.
```

Output from **Prose**:

```
At last the Gryphon said to the Mock Turtle "Drive on, old
fellow!  Don't be all day about it!" and  he  went  on  in
these words:--
"Yes,  we  went  to  school  in the sea, though you mayn't
believe it--"
"I never said I didn't!" interrupted Alice.
"You did." said the Mock Turtle.
     "Hold your tongue!" added the Gryphon,  before  Alice
could speak again.
```

When you use one of these methods to create a paragraph, **Prose** only does a justification break. **Prose** will not skip lines or indent unless blank lines or indentations explicitly appear in the input file. You can do fancier things with the .PARAGRAPH directive, to be introduced later.

A General Look at Directives

In its default mode, **Prose** automatically fills and justifies output lines, formatting the output in pages. Directives instruct **Prose** to do anything fancier. Directives can change the margins, control options, and define the type of output device you intend to use.

A line of directives is indicated by the directive escape character in the first column of an input line. The period is the default directive escape character (although you can change it if you wish) because it seems unlikely that anyone will want to type a period in the first column of a line of text. Several directives can be typed on the same line, provided that they are separated by the directive escape character. For example:

```
.BREAK.SKIP 2.MARGIN( L5 R65 )
```

Some directives take the remainder of the line as their parameter, so no other directives can follow these. Long directives may extend to several lines. Continuation lines are indicated by a plus sign (+) typed in column one. The continuation may be made anywhere that a blank is allowed. For example:

```
.FORM( [ /// L58 // #73 'PAGE' p /// ]
+      [ /// L58 //     'PAGE' p /// ] )
```

Examples in this guide will show directives typed in upper case, but both upper-case and lower-case characters may be mixed.

Every directive begins with the name of the command, MARGIN for instance. The name can be abbreviated to three letters (in fact, **Prose** only examines the first three letters.) The directive name may be followed by a parameter. In the absence of a parameter, default values are used. The parameter has four forms:

1) The absence of any parameter.
2) A single numeric value.
3) The remainder of the directive line.
4) A information enclosed in parentheses, consisting of descriptors defined by the directive itself.

When a numeric value is required (for a parameter or as part of a descriptor), an explicit positive integer may be given. In many directives, a relative value may be used. 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 number of the integer.

In the following example, the left margin is set to 10 and the right margin to 70. Then, the margins are squeezed together by 5 characters on both sides.

```
.MARGIN( L10 R70 )
.MARGIN( L+5 R-5 )
```

Controlling the Formatting Environment

The formatting environment is defined to be all the options and specifications that direct **Prose** as it produces formatted output from unformatted input. The concepts that make up the formatting environment can be loosely grouped into six areas; directives control each one:

1) INPUT controls the meaning and treatment of characters on the input file.
2) OUTPUT describes the type of output device for which the formatted result is intended.
3) FORM specifies the format of the page into which the running text will be inserted. This includes information on the placement of titles, footers, and the like.
4) MARGIN sets the left and right margins.
5) PARAGRAPH describes special actions for the beginning of each paragraph.
6) OPTION controls the rest of the miscellaneous options that affect the text formatting process.

Of these six groups, the INPUT, MARGIN, OPTION, and PARAGRAPH settings are likely to be changed often throughout the test. The number of different settings, however, will probably be small. To accommodate these needs, a simple device is available for these four directives.

The setting of options controlled by these directives follows this syntax:

```
.directivename( parameters )
```

where the parameters consist of a key letter followed by option settings. For instance:

```
.MARGIN( L5 R60 )
```

sets the left margin to 5 and the right to 60. Each time one of the four directives is processed, **Prose** saves the new values in a **keep buffer**. Ten keep buffers (numbered 0 through 9) are associated with each directive. A keep parameter may be used to specify the buffer to be used; if not specified, the values are saved in the numerically next buffer.

Old values may be recalled by using the following form:

```
.directivename number
```

For example:

```
.MARGIN 5
```

sets the margins to the values stored in keep buffer 5.

If no parameter is specified, the values are set to those stored in the numerically previous keep buffer. Since the keep number is automatically incremented when the parenthesis form is used and automatically decremented when no parameter is given, the keep buffers can be used as a stack.

```
.MARGIN( L0 R70 )

...

.MARGIN( L10 R60 )

...

.MARGIN
```

In the previous example, the last MARGIN directive resets the margins to their previous values: left 0 and right 70.

Summary Directive Table

Directive	Meaning (action)	Break	Parameter type
BREAK	break justification	*	–none–
COMMENT	no action		remainder of line
COUNT	set page count		numeric
FORM	define page format	*	(...)
INDENT	indent following line	*	numeric
INPUT	set input parameters	*	(...) or numeric
INX	store index entry		remainder of line
LITERAL	print literal text		remainder of line
MARGIN	set margins	*	(...) or numeric
OPTION	set options	*	(...) or numeric
OUTPUT	set output parameters		(...)
PAGE	eject to top of page	*	numeric
PARAGRAPH	set paragraphing params		(...) or numeric
RESET	reset directive defaults	*	(...)
SELECT	select pages to print	*	(...)
SKIP	skip output lines	*	numeric
SORTINDEX	sort and print index	*	(...)
SUBTITLE	set the subtitle		remainder of line
TITLE	set the main title		remainder of line
UNDENT	undent following line	*	numeric
WEOS	write end of section	*	–none–

The directives marked with an asterisk (*) cause a justification break before they are processed, since they affect the filling and justifying environment.

(...) indicates that the parameter is enclosed in parentheses and is described in detail along with the description of the directive itself.

Details on Directives

BREAK

Causes a justification break

COMMENT

Prose treats the remainder of the directive line as a comment; i.e., it is ignored. The COMMENT directive allows you to include information in the source of your document that will not be printed in the formatted copy.

COUNT, COUNT number

Sets the page counter. The numeric parameter can be relative; for example, `.COUNT +1` increments the page number by one. Without a parameter, the default sets the page number to one.

FORM, FORM (parameters)

Defines the page format, including titles, footers, date/time, and the top and bottom of the page. The argument consists of parameters, followed by an optional field width, if appropriate. For example, `T:30` prints the title in a field of 30 characters. Text lines are built by the FORM directive from left to right, starting in the first printable column, although the tabbing specification may be used to alter that. The following table describes the available `FORM` specifications.

Key Char	Meaning	Default Field Width
C	24 hour clock as `hh.mm.ss` (15.37.58)	8
D	raw date as `yy/mm/dd` (78/02/13)	8
E	nice date as `dd Mmm yy` (13 Feb 78)	9
Ln	fill in **n** lines of running text	
Pf	current page number, **f** selects the form:	3
	N or n arabic numerals (default)	[the field
	L upper case letter	width will
	l lower case letter	be expanded
	R upper case roman numerals	if needed]
	r lower case roman numerals	
S	subtitle	its length
T	main title	its length
W	wall clock as `hh:mm AM` (3:37 PM)	8
	or `hh:mm PM`	
#n	tab forward or backward to absolute column **n**	
'...'	print literal text	
/	print an end of line (by itself, a blank line)	
/n	print **n** ends of lines	
[	define top of page	
]	define bottom of page	

Default form (PDP-11)

```
.FORM( [ // T #62 E /// L54 /// #33 '- ' PN:1 ' -' //// ] )
```

The FORM directive is processed interpretively. The FORM argument is re-scanned as each page of output is produced, so that a change in a title buffer with the TITLE or SUBTITLE directive will change the title or subtitle on the next page.

The top of page definition is used for several things. By using the OUTPUT directive, you can request **Prose** to send a page eject to the output device when it reaches the top of a page. You also can request **Prose** to pause at the top of each page to allow you to change paper. At the end of the document, **Prose** does one last page eject and continues to interpret the FORM specification until it reaches the top of page. (This ensures that any commands specified for the bottom of the last page — such as a page number — are executed.)

Prose increments the page number at the bottom of the page specification, so if you print the page number both before and after the bottom of page definition you will get different numbers.

Once you understand the FORM directive, you can easily produce fancy page formats. You can, for example, design a FORM that will print the page number at the right of odd numbered pages and at the left of even pages. This is done with a FORM that defines two pages with two ['s and two]'s:

```
.FORM ( [ // T #62 E /// L56 // #63 'PAGE' P /// ]
+       [ // T #62 E /// L56 //     'PAGE' P /// ] )
```

In the absence of any of the parameters described above, no special page formatting is done. The result is a FORM consisting of a single L specification defining an infinite number of lines per page. In this mode, a PAGE directive with no parameter will put 5 blank lines between sections of text.

INDENT, INDENT number

Indents the following line by a certain number of spaces. In the absence of a parameter, the default is 5.

INPUT, INPUT number, INPUT (parameters)

The INPUT directive is used to define the input environment, that is, the interpretation of characters in the input file. The parameters, which can be given in any order, consist of a key letter followed by a value. The following table summarizes the parameters.

Key Letter	Meaning	Type	Default	Relative
B	explicit blank character	character	nul	
C	case-shift character	character	nul	
D	directive escape character	character	'.'	
H	hyphenation character	character	nul	
K	keep	number	next	no
U	underline character	character	nul	
W	input width	number	150	no

The default value is assigned when **Prose** begins processing. The default value also will be set if the key letter is given by itself. The value of a parameter changes **only** when a is given.

B: The explicit blank character indicates a blank that **Prose** should not tamper with. Thus, if the cross-hatch (#) is specified as the explicit blank:

```
.INPUT( B# )
```

then two words that are separated by an explicit blank:

```
Mr.#Smith
```

will never be split from one line to the next, and **Prose** will never fill blanks in between the words to justify a line.

C: The **case-shift character** must be used to create mixed-case output from upper-case-only input. When a case-shift character is specified, **Prose** automatically shifts all upper-case letters to lower case. To specify an upper-case letter, you must surround letters with the case-shift characters, causing a shift-up and shift-down. Since most upper-case letters are at the beginning of a word (following a blank), another method, called stuttering, is to double the first character of a word to achieve upper-case characters. The following example demonstrates the production of mixed-case output from upper-case-only input.

Input to **Prose**:

```
.INPUT(C^)
TTHE MMOCK TTURTLE WENT ON.
     "^W^E HADE THE BEST OF EDUCATIONS--IN FACT, WE WENT TO
SCHOOL EVERY DAY--"
     ^I'VE^ BEEN TO A DAY-SCHOOL, TOO," SAID AALICE. "^Y^OU
NEEDN'T BE SO PROUD AS ALL THAT."
     "^W^ITH EXTRAS?" ASKED THE MMOCK TTURTLE, A
LITTLE ANXIOUSLY.
     "^Y^ES," SAID AALICE: "WE LEARNED FFRENCH AND MUSIC."
     "^A^ND WASHING?" SAID THE MMOCK TTURTLE.
     "^C^ERTAINLY NOT," SAID AALICE, INDIGNANTLY.
     "^A^H TTHEN YOURS WASN'T A REALLY GOOD SCHOOL," SAID THE
MMOCK TTURTLE IN A TONE OF GREAT RELIEF. "^N^OW, AT ^OURS^, THEY
HAD, AT THE END OF THE BILL, '^F^RENCH, MUSIC, ^AND
WASHING--^ EXTRA.'"
```

Output from **Prose**:

```
The Mock Turtle went on.
     "We had the best of educations--in fact, we  went  to
school every day--"
     "I'VE  been  to a day-school, too," said Alice.  "You
needn't be so proud as all that."
     "With  extras?"  asked  the  Mock  Turtle,  a  little
anxiously.
     "Yes," said Alice: "we learned French and music."
     "And washing?" said the Mock Turtle.
     "Certainly not," said Alice, indignantly.
     "Ah Then yours wasn't a really good school," said the
Mock Turtle in a tone of great relief.  "Now at OURS, they
had, at the end of the bill, "French, music, AND WASHING--
extra.'"
```

At first glance, the stuttering method may seem clumsy, but experience shows that it is reasonably easy to get used to. To enter words that already have a double letter at the beginning (like llama and oops), merely precede the word with two case-shift characters, causing a shift-up/shift-down (^^LLAMA and ^^OOPS). The case-shift character does not need to be used unless you want to create mixed-case output from upper-case-only input. If possible, use mixed-case input to create mixed-case output.

D: The **directive escape character** is the character you type in the first column of an input line to flag it as a directive line.

H: The **hyphenation character** is used to define hyphenation points within words. Sometimes a long word will cause many blanks to be inserted to justify the preceding line. **Prose** will

hyphenate 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 set to be the slash (/), you may type "syncopation" as "syn/co/pa/tion". **Prose** will insert a hyphen (-) only when the characters on both sides of the hyphenation point are letters. You may type "hyper-active" as "hyper-/active", and **Prose** will split the word if necessary, without adding a superfluous hyphen. If **Prose** is forced to insert more blanks than a certain threshold (set with the OPTION directive), it will issue a message suggesting that you insert hyphenation characters.

K: The **keep** parameter explicitly specifies the keep buffer to be used to store the new input options. The default is for **Prose** to use the numerically next buffer.

U: Text surrounded by the **underline character** will be underlined. Blanks are not underlined, but explicit blanks are.

W: The input width is used to specify the number of characters to be read from each input line. If your input lines have sequencing information at the right of each line, you will need to set the width to an appropriate value.

INX text

Enters the remainder of the line together with the current page number as an index entry. If the formatted text migrates from page to page, the resulting index will always be correct.

LITERAL text

Prints the remainder of the line on the output file. The special processing for upper/lower case, underlining, and literal blanks is performed on the text of the parameter, and then it is printed as a single output line. This output line is printed independently of filling and justifying and page formatting processes; it is transparent to the usual **Prose** formatting and is not counted as an output line. The LITERAL directive is useful for producing special printer control characters on some systems. LITERAL is of little use on the PDP-11 and is described for the sake of completeness.

MARGIN, MARGIN number, MARGIN (parameters)

The MARGIN directive sets the left and right margins for filling and justifying. The **left margin** is the number of leading spaces before the first printed character, and the **right margin** is the column number of the last printed character. Thus, subtraction of the left margin from the right margin gives the number of printed columns. The parameters, which may be given in any order, consist of a key letter followed by a value. The following table lists the parameters.

Key Letter	Meaning	Type	Default	Relative
K	keep	number	next	no
L	left margin	number	0	yes
R	right margin	number	70	yes

Prose assigns the default value when it begins processing. The default value also is set when the key letter is given by itself. A parameter value is changed only when a specification is given.

The **keep** parameter explicitly specifies the keep buffer to be used to store the new margins. The default is for **Prose** to use the numerically next buffer.

OPTION, OPTION number, OPTION (parameters)

Miscellaneous options that affect the text formatting process are gathered together in the OPTION directive. These options are summarized in the following table. For switch options, + is on and - is off.

Key Letter	Meaning	Type	Default	Relative
E	print error messages	switch	+	n/a
F	fill output lines	switch	+	n/a
J	justification limit	numeric	3	no
K	keep	numeric	next	no
L	left justify	switch	+	n/a
M	multiple blanks	switch	+	n/a
P	2 blanks after periods	switch	+	n/a
R	right justify	switch	+	n/a
S	spacing	numeric	1	no
U	shift to upper case	switch	-	n/a

Prose assigns the default value when it begins processing. The default value also is set when the key letter is given by itself. A parameter value is changed only when a specification is given.

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 the E option is set to E-.

F: Output lines are automatically filled and justified as described in the section "Basic Units of Text". If the **fill switch** is turned off, **Prose** will print the input lines as they are, without reformatting to fill up the output lines. In effect, a justification break is done after each input line.

J: In justifying the left and right margins of an output line, **Prose** has to insert blanks that are not explicitly on the input file. The **justification limit** controls the point at which **Prose** will attempt to hyphenate a word. If, for instance, the justification limit is three, then the hyphenation process will be invoked when **Prose** has to insert three blanks between any 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.

K: The **keep** parameter explicitly specifies the keep buffer to be used to store the new options. The default is for **Prose** to use the numerically next buffer.

L: R: The **left and right justify switches** work together to determine the kind of justification to be done. If both switches are on, output lines are justified to both the left and right margins. If both switches are off, the lines are centered between the two margins. If one is on and one is off, the result is one straight margin (either left or right) and one ragged margin. The following demonstrates these four options.

```
OPTION( L+ R+ ) :

"You couldn't have wanted it much," said Alice;
"living at the bottom of the sea."

"I couldn't afford to learn it," said the Mock Turtle
with a sigh. "I only took the regular course."

"What was that?" inquired Alice.
```

"Reeling and Writhing, of course, to begin with," the Mock Turtle replied; "and then the different branches of Arithmetic--Ambition, Distraction, Uglification, and Derision."

"I never heard of 'Uglification,'" Alice ventured to say. "What is it?"

The Gryphon lifted up both its paws in surprise. "Never heard of uglifying!" it exclaimed. "You know what to beautify is, I suppose?"

```
.OPTION( L- R-) :

        "Yes," said Alice doubtfully: "it
      means--to--make--anything--prettier."
```

"Well, then," the Gryphon went on, "if you don't know what to uglify is, you are a simpleton."

Alice did not feel encouraged to ask any more questions about it: so she turned to the Mock Turtle, and said "What else had you to learn?"

"Well, there was Mystery," the Mock Turtle replied, counting off the subjects on his flappers--"Mystery, ancient and modern, with Seaography: then Drawling--the Drawling-master was an old conger-eel, that used to come once a week: he taught us Drawling, Stretching, and Fainting in Coils."

```
.OPTION( L+ R- ) :
```

"What was that like?" said Alice.

"Well, I can't show it to you, myself," the Mock Turtle said. "I'm too stiff. And the Gryphon never learnt it."

"Hadn't time," said the Gryphon: "I went to the Classical master, though. He was an old crab, he was."

"I never went to him," the Mock Turtle said with a sigh. "He taught Laughing and Grief, they used to say."

"So he did, so he did," said the Gryphon, sighing in turn; and both creatures hid their faces in their paws.

"And how many hours did you do lessons?" said Alice, in a hurry to change the subject.

```
.OPTION( L- R+ )

"Ten hours the first day," said the Mock Turtle: "nine
                                    the next, and so on."

                "What a curious plan!" exclaimed Alice.

      "That's the reason they're called lessons," the
   Gryphon remarked: "because they lesson from day to
                                                  day."

This was quite a new idea to Alice, and she thought it
 over a little before she made her next remark.  "Then
          the eleventh day must have been a holiday?"

              "Of course it was," said the Mock Turtle.

"And how did you manage on the twelfth?" Alice went on
                                                eagerly.

"That's enough about lessons," The Gryphon interrupted
in a very decided tone.  "Tell her something about the
                                              games now."
```

M: If the **multiple blanks switch** is on, multiple blanks in the input file are considered to be significant. That is, if there are several blanks between two words in the input file, there will be at least that many in the output file. **Prose** may add more blanks during justification. If the switch is off, multiple blanks will be changed into a single blank.

P: The **2 blanks after periods** option will ensure that each period already followed by at least one blank will be followed by at least two blanks. **Prose** will not add blanks before justifying if there are already two. This makes for nice spacing in your final copy even if you are not careful about typing spaces in the original.

S: Set the **spacing** option to 2 or 3 to get double-spaced or triple-spaced output, respectively. (Default is 1.)

U: Since some output devices are not able to handle mixed-case files, you can shift all lower-case letters to upper-case letters by selecting the **shift to upper case** option. This option is also handy for printing an entire unit, such as a sample program, all in upper case.

OUTPUT (terminal-type parameters)

The OUTPUT directive defines important aspects of the output device to which you will send the formatted text. The OUTPUT directive may be used only once. It must appear in one of only two places: before any lines are printed on the output device or immediately after the .RESET(OUTPUT) directive.

Terminal-type may be one of the following.

(**PDP-11**) All of the following is specific to the PDP-11.

ASC ASCII terminal, uses back space for underlining, but is otherwise the same as LPT below. There is, however, a difference in the handling of pauses for page eject. (See the P option below.)

LPT Line printer, using overprinting with a carriage return to do underlining. This is the default output device.

VTR Video Terminal (DEC VT100 or VT52). This does underlining by means of the control sequences defined for these terminals. The parameters define further characteristics of the output device, along with several global output options. The parameters, which may be given in any order, are listed in the following table.

Key Letter	Meaning	Type	Default
E	page eject at top of page ([in FORM description)	switch	-
P	pause at top of page	switch	-
S	shift output lines to the right	numeric	0
U	underlining is available	switch	+

E: The **page eject** option will print a form feed each time a [is encountered in the FORM specification.

P: Selection of the **pause** option will cause **Prose**

to stop printing and await operator acknowledgement each time a [is encountered in the FORM specification. On an ASC terminal, **Prose** will sound the bell and wait for a carriage return to be entered. For an LPT output device, no action will be taken.

S: All output that **Prose** produces can be **shifted to the right** by any number of spaces up to 50. This makes it easy to center the output on a wide printer page.

U: If the destination terminal does not have underlining ability and your input does underlining, the **underlining available** option should be turned off to prevent **Prose**

from trying to generate overprinted underlines.

PAGE, PAGE number

Causes a page eject if there are fewer than the specified number of lines remaining on the current page. If no parameter is given, PAGE does an unconditional page eject.

PARAGRAPH, PARAGRAPH number, PARAGRAPH (parameters)

Paragraphs can be indicated by any of the methods introduced in the section "Basic Units of Text". The PARAGRAPH directive provides a more versatile method of creating paragraphs.

The PARAGRAPH directive specifies the action to be taken when a new paragraph is signaled by the **paragraph flag character** in the first column of an input line. An automatic indent or undent can be selected, an automatic skip and/or automatic page eject can be specified, and you can even have **Prose** automatically number the paragraphs.

Key Letter	Meaning	Type	Default	Relative
F	paragraph character	character	nul	n/a
I	Automatic indent	number	0	no
K	keep	number	next	no
N	number generator		none	n/a
P	automatic page eject	number	0	no
S	automatic skip	number	0	no
U	automatic undent	number	0	no

Prose assigns the default value when it begins processing. The default value also is set when the key letter is given by itself. A parameter value is changed only when a specification is given.

F: The **paragraph flag character** invokes this collection of paragraphing actions when it appears in the first column of an input line. Note that this character must be set in at least one PARAGRAPH directive, or none of these actions will work.

I: U: The **automatic indent** or **automatic undent** is applied to the first line of the paragraph (see the description of INDENT and UNDENT). If the number generator is used, the indent or undent is applied after the number is generated.

N: If the **number generator** is specified, a new number (or letter) will be generated for each occurrence of the paragraph flag character. The number generator is initialized to 1 each time new paragraph settings go into effect. Resumption of an old setting will also resume the old numbering. The number replaces the paragraph flag character when the line is formatted. The number generator parameter has the form: Nfn.

f selects the numeric form:

–blank–	no numbering
N or n	arabic numerals
L	upper-case letter
l	lower-case letter
R	upper-case roman
r	lower-case roman

n is the field width, which will be expanded if needed. If, for example, you want an arabic numeral with three spaces left for the numeral, the command is **Nn3**.

P: The **automatic page eject** simulates the effect of the directive

```
.PAGE  number
```

before the first line of the paragraph. If this parameter is set to 4, for instance, a page eject is done if fewer than four lines will be left at the bottom of the page for the paragraph. The command is applied after the automatic skip.

S: The **automatic skip** is done before the first line of the paragraph, and functions the same as a skip directive.

K: The **keep** parameter explicitly specifies the keep buffer to be used to store the new paragraph options. The default is for **Prose** to use the numerically next buffer.

RESET, RESET (parameters), RESET (EXCEPT parameters)

The RESET directive sets directives to their default values. If you have changed the values of many directives (such as FORM, MARGIN, or OPTION), the simple command

```
.RESET
```

resets the values of all directives to their defaults. You also may reset directives selectively by using the second form of the command. For example,

```
.RESET( MARGIN OPTION )
```

only resets the MARGIN and OPTION directives. Directives also may be excluded. For example,

```
.RESET( EXCEPT FORM OUTPUT )
```

resets all directives with the exception of FORM and OUTPUT.

Any of the following parameters may be reset:

COUNT	FORM	INPUT	INX
MARGIN	OPTION	OUTPUT	PAGE
PARAGRAPH	SELECT	SUBTITLE	TITLE

The values of parameters for most directives are set to their defaults (listed with the description of each directive), except for the **keep** parameters, which are set to KO. For the COUNT, INX, and PAGE directives, however, the action is different. A reset of COUNT sets the page counter to 1; a reset of INX deletes all index entries that have accumulated; and a reset of PAGE causes a page eject. In addition, a FORM or OUTPUT reset also causes a page eject, since these directly affect the printed result.

SELECT (parameters)

You may use the SELECT directive when you need to print only certain pages of a document. The entire text will be formatted, but only the selected pages will be printed. Processing time will not be reduced much, but printing time will be. The descriptor consists of page numbers separated by spaces. Two page numbers separated by a colon (:) will select the span of pages between the pages, inclusive. The second page number may be specified relative to the first. The following example selects pages 3, 5, 10 through 15, and 20 through 25 to be printed.

```
.SELECT( 3 5 10:15 20:+5 )
```

The default is to select all pages to be printed.

SKIP, SKIP number

Skips a certain number of output lines, i.e., prints blank lines. SKIP will never print blank lines at the top of a page, so to skip lines at the top of a page at least one actual blank line must precede the SKIP directive. The default of SKIP is 5 lines.

SORTINDEX, SORTINDEX (parameters)

Index entries accumulated by INX directives can be sorted either alphabetically or by page number, then printed in a relatively flexible manner. The SORTINDEX directive allows you to specify the column that is to be considered the first significant column for alphabetical sorting, the number of leading blanks to print at the left of each index line, and the way to format the page number. The parameters, which may be given in any order, and listed as follows.

Key Letter	Default	Meaning
L	2	Left width of page number (field width of number)
M	0	Margin (left margin before index line)
P	0	Column (in index entry) to insert page number
R	2	Right width of page number, blanks printed after
S	1	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.

In the absence of parameters, defaults are used.

SUBTITLE text

Enters the remainder of the directive line into the subtitle buffer. The subtitle buffer is used by the FORM directive.

TITLE text

Enters the remainder of the directive line into the main title buffer. The title buffer is used by the FORM directive.

UNDENT, UNDENT number

Undents the next line a certain number of spaces. The undent is sometimes known by the name "outdent" or "hanging indent". A line can never be undented past the leftmost column of the printer page, so a large number is adjusted to a smaller value. In the absence of a parameter, the default is to undent to the leftmost printable column.

WEOS

Write an end-of-section on the output file. This directive is useful for creating multiple section writeups under systems with utilities that manipulate multiple section files.

(**PDP-11**) This directive has no effect on the PDP-11 version of **Prose**.

Running Prose

After formatting the text as desired, invoke **Prose** as shown:

```
.R PROSE
* output-file = input-file
```

input-file: The **Prose** source file(s), with a default extension of .PRS. Multiple input files are read and concatenated in the order listed.

output-file: The formatted **Prose** file. If the output file and the = are omitted from the command line, the output file will take the name of the last input file and the extension .DOC. Such a default output file will be placed in the default directory. The default extension for the output is .DOC.

We recommend as a general practice that you set up each **Prose** text without OUTPUT or FORM directives. Instead, keep these directives in another file that you will use as the first input file.

For example, Oregon Software supplies two header files with the **Prose** utility. One file, VT100.PRS, contains a set of OUTPUT and FORM directives for terminals. The other, PRINTER.PRS, contains a set of OUTPUT and FORM directives for the line printer.

If these header files are stored on the system device, you would use this command to prepare the document for the terminal:

```
DOCNAM = SY:VT100,DOCNAM
```

and this command to prepare the document for the line printer:

```
DOCNAM = SY:PRINTER,DOCNAM
```

These header files may be edited as needed, or you may create your own.

Conversion

Contents

Conversion From Pascal–1 to Pascal–2

Introduction

This manual describes the significant differences between **Pascal–1** (V1.2) and **Pascal–2** (V2.0) and the steps for conversion of programs from **Pascal–1** to **Pascal–2**.

Pascal–1 is a one-pass compiler for the Digital PDP-11 series that contains several low-level extensions giving the programmer control over the hardware and operating system. **Pascal–1** generates assembly code with no attempt at global optimizations.

Pascal–2 is a transportable multi-pass compiler that emphasizes conformance to the Pascal standard while generating excellent code. Properly used, **Pascal–2** will allow programs to be transported between computer systems with a minimum of change; the compiler itself has already been installed on computers made by two different manufacturers, and it is currently being implemented on two others.

Pascal–2 generates code that is shorter and faster than **Pascal–1** code. In many cases, programs running with all run-time checking enabled will be smaller than the same program running without checking under **Pascal–1**.

Programs that are acceptable to **Pascal–1** may not be acceptable to **Pascal–2** for the following reasons: **Pascal–2** has tighter error checking than **Pascal–1**; extensions that are considered ill-advised or that are not practical in an optimizing compiler have been removed (in-line code, for instance); some extensions have been modified; certain improvements create compatibility problems; finally, **Pascal–2** conforms to recent changes and clarifications in the language definition as a result of the international standardization effort.

The conversion procedures given in this manual have two goals. The first is to get programs that work under **Pascal–1** to work under **Pascal–2**. The second is to eliminate or isolate machine dependencies so that the resulting programs can be moved to other machines or to other Pascal compilers with relative ease.

This conversion guide is our effort to pass on to users our own experience and ideas in converting programs from **Pascal–1** to **Pascal–2**. Warning: No rote conversion procedure will work. And you must read the **Pascal–2** Language Specification before attempting any conversions.

Differences Between Pascal–1 and Pascal–2

The following summary of the ways in which **Pascal–2** differs from **Pascal–1**, though not complete, describes most cases.

Improvement of Error Checking

This section does not enumerate all the places where the checking has been tightened up, but it does list the changes that you probably will encounter most frequently in converting programs from **Pascal–1** to **Pascal–2**.

1. **Pascal–2** does range checking at compile time, and assignment of a constant that is out of the range of the variable will be marked as an error. In particular, an attempt to assign a negative value to an extended-range (unsigned) integer will be detected as an error.
2. **Pascal–2** detects assignments of out-of-range values to subrange variables. An extended-range variable, however, may have the same bit pattern as a negative number, and there

is no way to tell at run time whether the value should be negative or extended. An error will be noted, however, if the value, considered as an extended-range value, is outside the range limits of the variable.

3. **Pascal–2** detects dereference of a `nil` pointer at run-time.

4. **Pascal–2** can, at compile time, detect many uses of uninitialized variables and will flag the uses as errors. For instance, the compiler will detect as an error the use of a `for` loop variable after exit from the loop.

5. **Pascal–2** flags as an error any attempt to assign a value to a `for` loop variable within the loop.

6. A file, or any structure containing a file, may not be assigned or passed as a value parameter to a procedure. **Pascal–1** allows the assignment of a value parameter, but actually assigns a pointer to the file control block, not the file.

7. A component of a packed structure may not be passed as a `var` parameter to a procedure. This is most likely to show up when a component of a `packed array of char` is passed as a parameter. This limitation does not apply to the standard procedure `read`.

8. **Pascal–2**'s checking for scope rules follows the Pascal standard, and you cannot can use a name in a scope and subsequently re-define it in that scope, as **Pascal–1** allows. (The **Pascal–1** usage probably results in an undetected error.)

Removal of Extensions

The following extensions have been removed.

1. The `exit` statement in **Pascal–1** has been removed from **Pascal–2**. `Exit` can be replaced with a `goto` statement.

2. The **Pascal–1** pre-declared type `alfa` does not appear in **Pascal–2**. It can be replaced with the type declaration

```
type alfa = packed array [1..10] of char;
```

3. Embedded assembly code is not allowed in **Pascal–2**. Most in-line code can be moved into MACRO subroutines that can be called from your **Pascal–2** program.

4. In **Pascal–1**, any `array of char` may be treated as a string. A **Pascal–2** string is more tightly defined as the standard:

```
packed array [1..n] of char;
```

   This one change will probably cause many conversion problems, but the cure is simple, and the modified program will run under both **Pascal–1** and **Pascal–2**.

5. The **Pascal–1** pre-defined function `float` does not exist in **Pascal–2**. It is redundant: `R := I` is equivalent to `R := float(I)`. You can eliminate `float` wherever it is used.

6. The pre-defined functions `log` and `exp10` have been removed; they can be defined in terms of `ln` and `exp`, as described in "Likely Error Messages and Countermeasures".

7. **Pascal–2** does not accept the non-standard comment brackets `/*` and `*/`. The PASMAT formatter will replace such comment brackets with standard brackets.

8. **Pascal–2** does not allow the following substitute characters, which were allowed in some early versions of **Pascal–1**:

Pascal–1 Symbol	Standard Symbol	Function
_ or ←	:=	assignment
#	<>	"not equal" comparison
&	and	logical "and" operator
!	or	logical "or" operator

Changes to Extensions

Changes have been made in the implementation of the following extensions.

1. External procedures are handled differently. **Pascal–1** external procedures are defined by a compiler switch and are called by the **external** directive. In **Pascal–2**, the **external** directive serves both functions. If the body of the external procedure appears in the module, then it is available for other modules to use (defined externally). If the body does not appear in the module, the procedure is assumed to be a reference to another module (an external reference).
2. The **fortran** directive of **Pascal–1** is renamed to the **nonpascal** directive of **Pascal–2**.
3. **Origin** variables can have addresses only in the system space (less than 1000B) or in the I/O space (greater than or equal to 160000B). (In **Pascal–1 origin** addresses can be anywhere.)
4. Only actual strings are allowed as the file name and default arguments for **reset** and **rewrite**. **Pascal–1** accepts character constants (such as 'a'), which it converts to string constants.
5. The **Pascal–2 ref** function replaces the **Pascal–1** use of '@' as an address operator.

Improvements to Implementation

1. Packing of structures is implemented and can result in major space savings in run-time data. The cost in unpacking code is usually small. Packing is discussed at greater length in the Programmer's Guide.
2. A record allocated on the heap by **new**$(ptr, tag_1, tag_2, ..., tag_n)$ is now allocated the exact amount of storage required for the variants specified. This can result in considerably less heap storage being used. The data must be disposed of by the corresponding form of dispose, and the record itself may not be assigned as a unit.
3. The standard procedures **pack** and **unpack** are now implemented. This change simplifies the transport of programs written on other systems, but the use of these procedures is not recommended in new code. A **for** loop is always more efficient and is usually easier to understand.
4. The procedures **read** and **write** now apply to all file types.
5. An extension allows the definition of structured constants. This is useful for constant tables or initialization.
6. A **loophole** function provides a simple and visible method of violating the usual type compatibility rules.
7. The **%include** lexical directive simplifies the use of header files or standard code.

Changes in Implementation Definitions

1. Sets must have a base type in the subrange of 0..255, or any enumeration type with 256 elements or less. This includes a full set of char.
2. The character set in **Pascal–2** is 128 characters (7-bit), and the character set in **Pascal–1** is 256 characters (8-bit). The difference can cause compile-time range errors on programs that set the parity bit. One solution is the declaration of a file of 0..255. This file will no longer be a text file, however, and no references can be made to `eoln`.
3. ASCII control characters, including tabs, are not allowed in string constants. Control characters may be written to a file with the `chr` function.

Changes to or Clarifications of Standard Pascal

Pascal–2 differs from **Pascal–1** in the following ways because of changes to or clarifications of the language standard.

1. `For` loop controlled variables must be simple variables, local to the routine in which the `for` loop is defined. In **Pascal–1** any variable can be used.
2. The compatibility rules for `var` parameters adhere to a more restrictive rule called "name compatibility". A variable passed as a `var` parameter must be declared with the same type identifier used to declare the parameter. **Pascal–1** rules are less restrictive.
3. A full procedure heading is now required for any procedure or function to be passed as a parameter, reducing the likelihood of type errors. See the Language Specification for details.
4. Though not yet part of an official standard, the `otherwise` clause in a `case` statement apparently will become a standard extension to Pascal. The `otherwise` has therefore been added to **Pascal–2** to serve the same function in a `case` statement as the `else` clause does in **Pascal–1**. **Pascal–2** still allows an `else` clause in the place of an `otherwise` clause, but `else` may not be allowed in a future release. The PASMAT text formatter can transform `else` to `otherwise`.
5. Line separators are not allowed in strings. The entire text of a string constant must be on one line. **Pascal–1** strings are allowed to span line separations, with a system-dependent line-separation character(s) inserted in the string. The change helps you find unterminated strings.
6. The standard now specifies that `input` and `output` are defined at the global level if used as program parameters. Since **Pascal–2** ignores program parameters, `input` and `output` are predefined as global variables in every program, even if they are not used. Thus, you cannot define the identifiers `input` and `output` at the global level as in **Pascal–1**.
7. The standard now specifies the "at" character '`@`' as an alternate character for the pointer symbol '`^`'.

General Procedures and the CONVRS Utility

Your first step in conversion is to install the **Pascal–2** library in place of the **Pascal–1** library. The **Pascal–2** library contains all of the routines in the **Pascal–1** library, plus additional routines used by **Pascal–2**. The **Pascal–2** library can be used by both compilers.

A utility program, CONVRS, is provided with **Pascal–2** releases to aid conversion. CONVRS scans your program, producing a list of potential problem areas. CONVRS will not detect problems that will be detected by the **Pascal–2** compiler. Most programs can be converted as described in the

following section, "Basic Conversion Techniques". CONVRS also will flag programs that use external procedures or functions as "external" and programs that use low-level techniques as "low-level". If your program is so marked, you must use additional conversion techniques described in "Programs Using External Procedures and Functions" and "Programs Using Low-Level Techniques".

CONVRS will flag the following problems, with an indication of where in the program they occur.

1. Embedded switches.
2. External routine definitions.
3. In-line code.
4. Use of variant records without tag fields.
5. Use of **origin**.

The CONVRS command line follows this form:

```
.R CONVRS
* output-file = input-file /switch
```

input-file: The form of the input file(s) should be identical to that used for a normal **Pascal–1** compilation.

output-file: This file will contain a list of possible incompatibilities, along with the file name, line number, and text of the line where the incompatibility was noted. The default extension for the output file is .CVR.

/switch: CONVRS needs the /E (external) compiler switch if it is required for the **Pascal–1** compilation; any other switches will be ignored.

For example, if the module is compiled with **Pascal–1** as:

```
.R PASCAL
*PROCS = HEADER,PROCS/X/E
```

CONVRS uses this command:

```
.R CONVRS
*PROCS = HEADER,PROCS/E
```

Basic Conversion Techniques

Programs contained in a single source file, using no externally compiled procedures and using no extensions such as in-line code, generally require little conversion effort. (Conversion of long programs, of course, may be time-consuming.) The main problems will probably arise from the improved error checking in **Pascal–2**.

If the program is split into multiple files, you may want to use the **%include** feature of **Pascal–2** to concatenate the files automatically at compilation. (See the Programmer's Guide for details.)

The conversion procedure follows.

1. Make sure that the program compiles under **Pascal–1**; double-check that CONVRS did not flag the program as "external" or "low-level".
2. Change any embedded switches noted by CONVRS to the equivalent **Pascal–2** switches. The following section, "Conversion of Embedded Switches", gives the equivalent **Pascal–2** switches.
3. Compile the modified program under **Pascal–2**, using the /list or /errors switch.

4. Referring to the listing and to "Likely Error Messages" described below, modify the program to eliminate the errors.
5. Repeat 3. and 4. above until the errors disappear.

At this point, you will have a clean compilation of your program. However, **Pascal–2**'s run-time diagnostics are considerably more detailed than those for **Pascal–1** and may detect additional errors. Apply standard debugging techniques to eliminate these errors.

Two changes in the heap management may cause the program to fail in unusual ways. Storage allocated on the heap is initialized to a system-dependent value (currently -1 but subject to change). If your program does not initialize a variable, your program may run differently than you expect.

Also, if a heap variable is allocated with **new**$(ptr, tag_1, ..., tag_n)$, only the amount of storage required to hold this exact variant will be allocated. If other variants are used, or if the record is assigned as a whole, you may get unexpected results at some point apparently unrelated to the assignment. These actions are illegal in standard Pascal, but there is no diagnostic for them.

Conversion of Embedded Switches

Embedded switches serve similar functions in both compilers, but the format and switch names differ. **Pascal–1** switches are single characters, followed by "+" or "-" to turn the switch "on" or "off". **Pascal–2** switches are full words. A **no** in front of the switch name turns off the switch. **Pascal–2** switches may be abbreviated to 3 characters. The CONVRS program will flag **Pascal–1** switches, and you must insert the equivalent switches for **Pascal–2**. The **Pascal–2** compiler will ignore any **Pascal–1** switches left in the code. The **Pascal–1** compiler, on the other hand, will accept (but may misinterpret) **Pascal–2** switches. (See the Programmer's Guide for details on switches.)

The following table gives the conversions:

`$A` Use `$arraycheck`. **Pascal–2** produces additional checks. See "Run-Time Checking" below.

`$C` No equivalent. Techniques for conversion of in-line code are described in "Programs Using Low-Level Techniques" below.

`$D` Use `$debug`. Note that this implies `$list` and that the Debugger is somewhat different for **Pascal–2**. (See the Debugger Guide for details.)

`$E` No equivalent. Techniques for conversion of external procedures are described in "Programs Using External Procedures and Functions" below.

`$F` Use `$sim`. Note that the handling of hardware options is different in **Pascal–2**; check the Programmer's Guide to determine the proper switches for your hardware configuration.

`$L` Use `$list`.

`$S` No equivalent. **Pascal–2** does not normally produce assembly code, and in an optimizing compiler no direct connection exists between the source lines and the code produced. In **Pascal–1**, `$S` is also used to modify the `$D` debugging command. Source-line debugging is always enabled in **Pascal–2**, so a **Pascal–1** program with `$S$D` is equivalent to a **Pascal–2** program with `$debug`.

`$T` Use `$stackcheck`. **Pascal–2** produces additional checks. See "Run-Time Checking" below.

`$X` Use `$double`.

Run-Time Checking

Pascal–2 supports more kinds of run-time checking than does **Pascal–1**. In addition to `$arraycheck` and `$stackcheck`, there are `$rangecheck` (assignment of a value to a subrange) and `$pointercheck` (use of a `nil` pointer). In many cases, the code generated by **Pascal–2** with all checking enabled is smaller than the code generated by **Pascal–1** without checks. You can (and should) leave checking turned on in most cases. However, `$nocheck` will generate the shortest and fastest program.

Programs Using External Procedures or Functions

Pascal–2 has a more general method of defining external procedures than does **Pascal–1**. The **Pascal–1** compiler switch (`/E` or `$E`) specifies that all procedures on the main program level are available to other modules (externally defined). The **Pascal–1** `external` directive declares that the module is using an external routine (an external reference).

In **Pascal–2**, the `external` directive serves both functions. The `external` directive is treated in a manner similar to the `forward` directive. If the procedure is defined when the `external` directive is given, the procedure is available to other modules (externally defined). If the external procedure is not defined, the procedure is assumed to be a reference to another module (an external reference).

Thus, you can place the declarations for all external procedures in a single header file that can be used to compile all modules.

For example, a **Pascal–1** external procedure is defined by:

```
{$E+ make the following procedure external}
procedure Proc(Arg: Argtype);
  begin
  {procedure body}
  end;
{$E-}
```

A **Pascal–2** external procedure is defined by:

```
procedure Proc(Arg: Argtype);
  external;

procedure Proc;
  begin
   {procedure body}
  end;
```

Note that with **Pascal–2** you may have external procedures in the main program. You may not need this feature specifically to convert to **Pascal–2**, but the new `external` declaration may simplify some programs.

Pascal–2 external procedures can be called from **Pascal–1** main programs or procedures. However, they cannot reference global variables or the files `input` and `output` unless the main program is compiled with **Pascal–2**.

The "/Pascal1" Compiler Switch

The `/pascal1` compiler switch causes the compiler to generate code for external calls, allowing you to call both **Pascal–1** and **Pascal–2** routines. If parameters or global variables passed between the two systems have different storage allocation, you may have difficulties. All global variables used and all global variables that textually precede the variables used must have the same storage allocation in both **Pascal–1** and **Pascal–2**.

The code generated for external procedure calls with this switch is about twice the size as that generated without the switch. It is to your advantage to eliminate the /pascal1 switch. (Even so, a **Pascal–2** program using the /pascal1 switch will still execute much faster than if the program remained totally in **Pascal–1**.)

The following is a list of differences in storage allocation.

1. **Pascal–1** compilers prior to revision V1.2H allocate extra memory for **text** files.
2. Packed structures (other than **packed array of char**) are allocated differently.
3. An enumeration type with less than 128 members is allocated a single byte of storage in both systems. An enumeration type with 129 through 256 members (inclusive) is allocated one byte in **Pascal–2** and one word in **Pascal–1**. Enumeration types with more than 256 members are allocated full words in both systems.
4. Sets are incompatible between the two systems.

Conversion Procedures

The exact procedures depend on the form of the program being converted. The following cases are of interest.

1. No global variables are used by the external procedures, or all global variables have compatible storage allocation — and — no incompatible parameter types exist. In this case, the /pascal1 switch can be used to convert the program in easy stages.
2. Some variables have incompatible storage allocation, either in the shared global section, or when passed as parameters. Any portions of the program having these problems must be converted as a unit.
3. Some of the procedures use low-level techniques. Isolate these procedures and call them using the /pascal1 switch until they can be converted as described in the next section.

The general procedure for conversion is:

1. Convert the main program to **Pascal–2** if at all possible. If not, you can convert only procedures that do not have global references.
2. Convert the external modules one at a time. If possible, use the /pascal1 switch to allow the conversion of one module at a time, then check each module for errors as it is converted to **Pascal–2**. This can considerably simplify the checkout procedure. Don't forget to remove any dummy main program in external modules to be compiled with **Pascal–2**. You also may want to include the $nomain switch in external modules.
3. Convert to the new form of external definition for external procedures to be converted. If all procedures are to be converted, you can move all external definitions into a header file.
4. When all modules have been converted, check the program modules as a whole as described under "Basic Conversion Techniques" and correct any errors discovered by the improved **Pascal–2** run-time checking.

Programs Using Low-Level Techniques

Pascal–2 has low-level extensions to provide direct interface to an operating system or to machine resources. **Pascal–2** also provides low-level constructs that will be detectable if the program is transferred to another system.

Unfortunately, many of the low-level programming techniques used in **Pascal–1** code are not directly detectable by the **Pascal–2** compiler. For instance, **Pascal–2** treats in-line code as a comment; that

part of the program will be omitted without warning. CONVRS attempts to flag in-line code; at times, however, CONVRS can only guess at what is going on, and it may flag some legitimate constructs or miss some low-level constructs.

Low-level techniques considered in this section are:

1. In-line assembly code.
2. The use of variant records to do type conversion.
3. `Origin` variables.

In-line Assembly Code

Here are three basic approaches to conversion of programs using in-line assembly code.

1. Recode the assembly code. This is frequently possible with the low-level extensions of **Pascal–2**.

If the in-line code is being used to insert .TITLE or a similar MACRO directive, read the section in the Programmer's Guide on the run-time environment to see whether you can achieve the same effect without the code insertions.

2. Code an external routine in assembly to do the function required. This is usually easy to do with the PASMAC utility macros provided with **Pascal–2**. (See the Utilities Guide for details.)
3. Leave the procedure using in-line code in **Pascal–1**, then use the `/pascal1` switch in all **Pascal–2** modules that use the procedure. Performance for the entire program will be reduced, but this approach may be the most practical.

Whatever approach is taken, no rote procedure will work. You will have to look at the purpose of the code and decide how to create the same effect using the **Pascal–2** tools provided. (See the Programmer's Guide for details.)

Variants For Type Conversion

Where variant records are used, with or without tag field, to do type conversion, you may have no difficulty converting from **Pascal–1** to **Pascal–2**.

The major difficulty, if it appears, will be the difference in field allocation between the two versions of Pascal. This difference can be remedied only with care and study of the declarations of the record types. You might use the `loophole` function instead of this method, as the compiler will then check for size compatibility and the non-standard action will be noted in an obvious way instead of being buried in standard-appearing code.

Additional difficulties may occur if you assign an absolute address to a pointer that in turn is used to access memory for some purpose. If the memory being accessed is in the I/O page, the value may be changed without action on the part of the **Pascal–2** program. The **Pascal–2** compiler may have eliminated some references to the locations as unnecessary, since it may preserve a local register copy of the value. Unexpected results are possible. (This is not a problem with `origin` variables.)

Solution: Make discrete references through a separate procedure, since the compiler does not optimize across procedure boundaries.

"Origin" Variables

Pascal–2 variables may be specified with origins in the system space (addresses less than 1000B) or in the I/O page (addresses greater than 160000B). If variables with origins outside that range are being used for communication with other processes, the code must be changed to use a pointer that has been set to the address of the variable. This pointer value can be set with the **loophole** function.

Likely Error Messages and Countermeasures

This section lists error messages that you are likely to see when compiling programs that compile without error messages on **Pascal–1**. For each message, likely causes are listed, plus solutions. The errors are listed in numerical order to make reference easier.

`*** 1: Illegal character`

A. A substitute character allowed in early versions of **Pascal–1** appears.

Solution: Replace it with the appropriate standard symbol.

B. A control character appears in a string constant.

Solution: Reprogram to use the **chr** function instead.

`*** 17: Block must begin with LABEL,CONST,TYPE,VAR,PROCEDURE,FUNCTION, or BEGIN`

A. A **fortran** procedure or function appears.

Solution: Change it to **nonpascal**.

`*** 45: Operand expected`

A. This is an "at" character (**@**) used as an address operator.

Solution: use the **ref** function instead.

`*** 58: Identifier cannot be redefined or defined after use at this level`

A. An identifier from an outer level has been used at the current level, then an identifier has been defined at the local level with the same name.

Solution: Find the earlier identifier (it is almost certainly a constant or type) and either change the use or change the name of one of the identifiers (normally, you'll change the local identifier).

B. The identifier is **input** or **output** and is declared at the main program level. **Pascal–2** makes these identifiers pre-defined at the main program level even if they are not used.

Solution: Change the name of the offending identifier.

`*** 62: Undefined identifier`

A. If this is an **exit** statement, either change the loop logic to remove the need for the **exit**, or replace it with a **goto** statement to a label beyond the end of the loop. Remember to declare the label in the declarations for the procedure.

B. If this is the type **alfa**, which is predefined in **Pascal–1**, insert the global declaration `alfa = packed array [1..10] of char;`.

C. If this is the **Pascal–1** predefined function **float**, remove it, or define a **float** function:

```
function float (I:integer): real;
begin float := I; end;
```

D. If this is a **Pascal–1** predefined function `log` or `exp10`, write a function in Pascal to do the equivalent, using the identities:

$$log_{10}(x) = ln(x)/ln(10) = 0.43429448190325183*ln(x)$$

$$exp_{10}(x) = exp(x*ln(10)) = exp(2.30258509299490457*x)$$

E. An assignment with the **Pascal–1** special character '_' is considered to be an undeclared identifier in **Pascal–2**, which allows the '_' character in an identifier.

Solution: Change the form of `a_b` to `a := b` for assignment.

*** 76: `Reassignment of FOR-loop control variable not allowed`

A. The controlled variable of a `for` loop is the target of an assignment statement, or is being passed as a `var` parameter. This is not checked in **Pascal–1**.

Solution: If the assignment is to terminate the loop prematurely, change the loop to a `while` or `repeat` statement. If the assignment is an error, correct it.

*** 89: `Too many actual parameters`

A. This is a call to a procedure passed as a parameter. In **Pascal–1**, no parameter specifications are provided in the declaration of the procedure parameter, so **Pascal–2** assumes that this is a procedure without parameters.

Solution: Specify the parameters in the declaration.

*** 91: `Actual parameter type doesn't match formal parameter type`

A. If the actual parameter is a string, the formal parameter is not declared `packed array [1..n] of char`.

Solution: Change the declaration to correspond to the standard rules. Note that if the original declaration had a lower bound other than 1, some code may have to be changed as well.

B. If this is a procedural parameter, the parameter declaration must be modified as specified under the error "Too many actual parameters", above.

*** 95: `Illegal comparison of record, array, file, or pointer values`

A. The operands being compared are of type `array of char`, and the standard will only allow comparison of `packed array [1..n] of char`.

Solution: Change the declarations as appropriate. If the previous declaration did not have a lower bound of 1, the program code may have to be changed as well.

*** 97: `Assignment operands are of differing or incompatible types`

A. You are trying to assign a string to a variable that is not of type `packed array [1..n] of char`.

Solution: Fix the declaration to be of the form `packed array [1..n] of char`. If the previous declaration did not have a lower bound of 1, the program code may have to be changed as well.

*** 107: `FOR-loop control variable must be declared at this level`

A. The Pascal standard now allows only local variables in a `for` loop.

Solution: Declare a local `for` control variable.

```
*** 115: Variables of this type are not allowed in WRITE
*** 119: Variables of this type are not allowed in READ
*** 121: Packed array [1..n] of characters expected
```

A. The variable is not of a string type.

Solution: Change the declaration to the form **packed array [1..n] of char**. If the previous declaration did not have a lower bound of 1, the program code may have to be changed as well. Note that single-character literals (such as `'a'`) are not strings.

```
*** 134: Must assign value before using variable
```

A. The variable must be assigned a value before being used.

```
*** 136: Assignment value out of range
```

A. A constant value assigned to a subrange variable is out of the subrange.

Solution: Correct either the assignment or the declaration of the variable.

```
*** 140: Files must be passed as VAR parameters
```

A. **Pascal–1** allows files to be passed as value parameters, but actually passes a pointer to the file, so it is the equivalent of a **var** parameter. **Pascal–2** conforms to the standard by not allowing files to be passed as value parameters.

Solution: Change the parameter declaration to be a **var** parameter.

```
*** 141: Assignment of file variables not allowed
```

A. **Pascal–1** allows the assignment of file variables, but it actually assigns a pointer to the low-level file control data structure. If this is done to generate a pointer to the low-level file control structure, the program uses low-level techniques; consult the appropriate section above. Otherwise, you must decide what is actually being done with this assignment and reprogram to avoid it. You might, for instance, have to pass the file as a parameter to the procedures using it.

```
*** 142: String constants may not include line separator
```

A. Under the new standard, a string constant may not span line boundaries. In **Pascal–1**, a system-dependent line separator character is inserted in the string.

Solution: Change the program to avoid the problem.

```
*** 143: Set types must have a base in the range 0..255
```

A. **Pascal–1** limits set types to 64 elements, and the lower bound can be any non-negative value. **Pascal–2** limits sets to base types with ordinal values within the range 0..255.

Solution: Change the program so the base type is within the allowed limits.

```
*** 145: Non-standard comment form, please use "{" or "(*"
```

A. Nonstandard comments of the form **/* ... */**.

Solution: Change to standard comment brackets. The PASMAT formatter will do this for you automatically.

Installation

Contents

Pascal–2 V2.0/RT-11 Installation

To install **Pascal–2**, V2.0 for RT-11 V3 and V4, follow the steps below:

1. Copy all of the **Pascal–2** files to the system device (SY:);
2. Select a compiler depending on your system monitor;
3. Select a run-time library depending on your processor hardware options;
4. Compile the **Pascal–2** utility programs;
5. Delete files no longer needed (optional).

These steps are described in detail in the following paragraphs, and are illustrated by an example that can be found in Appendix B of this Installation Guide. At the completion of installation, the **Pascal–2** system will be fully operational as described in the User's Guide.

Copying the Pascal–2 Files to the System Device

Copy all of the distribution files to the system device (SY:) using the PIP program. This will speed the installation process if there is sufficient disk space available.

```
.R PIP
*SY: = MT0:*.*                    replace MT0: with your distribution medium
```

If your system disk is a floppy disk, or is otherwise limited in available space, you should first read the following sections and select the files that will be necessary for your system. The minimal system requires one compiler file and one library file. You may wish to build more than one system disk and, for example, install the compiler, library, and Debugger on one disk and the utility programs on a second system disk.

After selecting the necessary files, you can examine the directory listings supplied with the **Pascal–2** system and copy only the required files to your system.

Selecting a compiler for your system monitor

There are two compilers supplied with **Pascal–2**. Your choice of a compiler depends on the version of the RT-11 monitor you will use. There are four possibilities: the Base-Line (BL) monitor, the Single-Job (SJ) monitor, the Foreground-Background (FB) monitor, and the eXtended-Memory (XM) monitor.

If you will be using either the BL or the SJ monitor, choose the compiler called SJ.SAV. If you use the XM monitor, select the compiler called XM.SAV.

The FB monitor does not leave sufficient memory to run the **Pascal–2** compiler. If you are using the FB monitor, you will need to switch to either the SJ or XM monitors (using the BOOT command) before compiling a Pascal program. Once compiled, programs may be linked and run under the FB monitor.

The XM compiler should be chosen over the SJ compiler where possible because it uses the virtual overlay capability and will give faster compilations.

When you have selected a compiler file, copy it to SY:PASCAL.SAV. This compiler will operate as described in the User's Guide. You may then delete SJ.SAV and XM.SAV, or you may leave them on your system disk for use under their respective monitors.

Selecting a run-time library

There are four run-time libraries supplied with **Pascal–2**, one for each combination of processor instruction sets. Choose the library that matches the configuration of the processor that will run your compiled programs.

The possible configurations are:

- FPP — a processor with the Floating Point Processor instruction set. The FPP is standard equipment on the PDP-11/60, and optional on all new PDP-11's and the LSI-11/23. If your processor includes the FPP, select the `LIBFPP.OBJ` library.
- FIS — the Floating Instruction Set. The FIS hardware is an option available for the LSI-11, LSI-11/2, and some older PDP-11 processors. If your processor has FIS, select the `LIBFIS.OBJ` library.
- EIS — Extended Instruction Set, for hardware support of multiply, divide, and long shift instructions. EIS is standard equipment on all new PDP-11 and LSI-11/23 processors, and an option available for all older LSI-11's and PDP-11's. If your processor has neither FPP nor FIS, but does have EIS, then select the `LIBEIS.OBJ` library.
- For processors with no extended or floating instructions, select the `LIBSIM.OBJ` library. This library will operate on any LSI-11 or PDP-11 regardless of its actual configuration, but will not take advantage of any optional hardware.

After selecting a library file, copy it to `SY:PASCAL.OBJ` to be used as described in the User's Guide. You may then remove the other library files, or leave them on the system for use with other configurations.

Compiling the utility programs

Six utility programs are supplied in source form. Each of the utilities should be compiled following the procedure in Appendix B. The utility programs will then be available for use as described in the Utilities Guide.

Installing Pascal–2 With Pascal–1

The **Pascal–1** system compiler can be renamed to allow simultaneous use of **Pascal–1** and **Pascal–2**. Then, the files for both systems can be present on the system device without conflict.

The object library `PASCAL.OBJ` can be shared by the two systems; the libraries supplied with **Pascal–2** include all of the **Pascal–1** routines.

For versions of **Pascal–1** prior to revision V1.2H, a change in the **Pascal–2** run-time library may affect existing **Pascal–1** programs: **Pascal–1** global variables are now initialized to bit patterns of all ones (minus one). Prior to V1.2H, global variables were not initialized but were often zero. Programs that have uninitialized variables are now more likely to fail. This helps to identify latent problems with uninitialized variables. **Pascal–1** programs that worked, but fail when linked with the **Pascal–2** libraries, should be carefully examined for variables that are used before acquiring a value.

Appendix A — Pascal–2 System Distribution Files

Compilers

SJ .SAV	**Pascal–2** Compiler for SJ monitor
XM .SAV	**Pascal–2** Compiler for XM monitor

Object Libraries

LIBFPP.OBJ	Library for processors with FPP and EIS
LIBFIS.OBJ	Library for processors with FIS and EIS
LIBEIS.OBJ	Library for processors with EIS only
LIBSIM.OBJ	Library for base-level processors
VIRJOB.OBJ	Header module for XM virtual jobs

Debugger and Profiler Modules

DBRUN .OBJ	Debugger root module
DBUSER.OBJ	Debugger user load module
DBUG .OBJ	Debugger overlay module
PRFILE.OBJ	Profiler

Utility Programs

PASMAT.PAS	Program formatter
PB .PAS	Pascal Beautifier
XREF .PAS	Cross-referencer
PROCRE.PAS	Procedure cross-referencer
PROSE .PAS	Text formatter
CONVRS.PAS	**Pascal–1** conversion aid
ERROR .PAS	System Error procedure
STRING.PAS	String package
PASMAC.MAC	MACRO-11 interface package

Documentation Files

INTRO .PRS	Introduction
USER .PRS	User's Guide
GUIDE .PRS	Programmer's Guide
PASCAL.PRS	Language Specification
DEBUG .PRS	Debugger Manual
UTILIT.PRS	Utilities Guide
CONVER.PRS	Conversion Guide
INSTAL.PRS	Installation Guide
VT100 .PRS	VT100 header
PRINTE.PRS	Line printer header

Demonstration Programs

HEARTS.PAS	Four-handed game of Hearts
LIFE .PAS	The game of Life
CHECKR.PAS	A Checkers game
PL0 .PAS	PL/0 – a simple compiler
RANDOM.PAS	Random number generator
MAZE .PAS	Amazing Demonstration

Appendix B — Sample Installation Command File

The following steps illustrate the installation of **Pascal–2** from magtape on a processor that includes the FPP floating point processor. The **Pascal–2** compiler for the single job (SJ) monitor is selected. The commands below will work for both V3 and V4 RT-11 systems.

```
.ASSIGN  SY:  DK:          ------------------------ assign SY: as the default device
.R PIP
*SY: = MT0:*.*             ------------------------ copy all files to the system disk
*PASCAL.OBJ = LIBFPP.OBJ   ------------------------ select the FPP run-time library
*PASCAL.SAV = SJ.SAV       ------------------------ select the SJ compiler
*^C

                           ------------------------ Build the utility programs

.R PASCAL                  ------------------------ Build the PASMAT formatter
*PASMAT
.R LINK
*PASMAT = PASMAT,SY:PASCAL
*^C
.R PASCAL                  ------------------------ Build the PB formatter
*PB
.R LINK
*PB = PB,SY:PASCAL
*^C
.R PASCAL                  ------------------------ Build the XREF cross-referencer
*XREF
.R LINK
*XREF = XREF,SY:PASCAL
*^C
.R PASCAL                  ------------------------ Build the PROCREF cross-referencer
*PROCREF
.R LINK
*PROCREF = PROCREF,SY:PASCAL
*^C
.R PASCAL                  ------------------------ Build the PROSE text formatter
*PROSE
.R LINK
*PROSE = PROSE,SY:PASCAL
*^C
.R PASCAL                  ------------------------ Build the CONVRS Pascal–1 conversion aid
*CONVRS
.R LINK
*CONVRS = CONVRS,SY:PASCAL
*^C

.R PIP                     ------------------------ Clean up the system disk (optional)
*SJ.SAV,XM.SAV/D           ------------------------ Remove the compilers, leaving PASCAL.SAV
                           ------------------------ Delete the libraries, leaving PASCAL.OBJ
*LIBFPP.OBJ,LIBFIS.OBJ,LIBEIS.OBJ,LIBSIM.OBJ/D
                           ------------------------ Delete the utility compilation files
*PASMAT.OBJ,PB.OBJ,XREF.OBJ,PROCREF.OBJ,PROSE.OBJ,CONVRS.OBJ/D
*^C
```