OMSI PASCAL-1™

Version 1.2 for RSX

Oregon
Software

Oregon
Software

2340 S.W. Canyon Road
Portland, Oregon 97201
(503) 226-7760
TWX 910-464-4779

March 9, 1980
Printed in USA

OMSI Pascal-1 V1.2/RSX User's Guide

Welcome to OMSI Pascal!

This is the introductory manual, 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 manual assumes that you are familiar with

1) simple RSX commands;

2) a text editor (EDIT, TECO, EDT, SOS);

3) elementary Pascal programming.

This manual is not:

1) an introduction to Pascal;
   (see Programming in Pascal by Grogono)

2) a detailed description of OMSI Pascal-1
   (see the Language Specification)

3) an expert's guide to programming with OMSI Pascal-1
   (see the Programmer's Guide)

So you want to run a Pascal program?

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

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

Source programs should be stored in files with the extension .PAS for Pascal (example: FIRST.PAS). The .PAS extension may be omitted from commands to the Pascal compiler, but must be included in commands to other RSX systems such as the editor.

After editing, your program must be compiled -- translated into a form which can be directly executed by the computer. The Pascal compilation process is directed by the 'PAS' system task. The Pascal compiler produces a .MAC assembler file; this is assembled using MACRO to produce a relocatable object file. The Task Builder combines the object file with the Pascal library to produce an executable task image. The entire compilation process requires these commands:

```
>PAS <file.MAC> = <file.PAS>
>MAC <file.OBJ> = <file.MAC>
>TKB <file.TSK>/FP/CP = <file.OBJ>,[1,1]PASLIB/LB
```

To illustrate the compilation process, assume that this program

```
program First (Output);
begin
   Write ('"Things are best in their beginnings"');
   Writeln (' - Blaise Pascal');
end.
```

is stored in the file FIRST.PAS. The compilation process proceeds as follows:

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

Notice the /FP and /CP switches in the Task Builder command, which enable saving Floating Point context and CheckPointing respectively. These switches are strongly recommended for Pascal tasks -- see the Programmer's Guide for the details.

Notice also that no errors were detected. This is certainly unusual if this is your first program! What happens if there are detectable errors in the source program?

The following program contains a deliberate error:

```
program Second (Output)
begin
   Writeln ('"Things get worse as they continue"');
end.
```

This program is missing a semicolon between the program heading and the keyword 'begin'. Semicolon errors are the most common errors made by beginning Pascal programmers. Semicolon errors are always detected by the compiler:

```
>PAS SECOND=SECOND
   2                        begin
                            ^
******          Expected 'SEMICOLON' missing

Errors detected:  1
Free memory:  6106  words
```

For each error, a line of the source program is printed followed by an arrow indicating the approximate position of the error, and a message describing the error. Many compilation errors are possible -- see Appendix A of the Programmer's Guide for a complete list.

As is often the case, we need to see more of the program to determine the precise location of the error and to correct it. The Pascal compiler can be directed to display the entire program, with all detected errors and other information. This is called the 'listing' of the program.

The second output file (if present) is the listing file, with the .LST default extension. For a listing at your terminal, specify 'TI:' as the listing file; the listing may also be written to the line printer or a disk file.

```
PAS <file.PAS>,<file.LST> = <file.PAS>
```

A listing of a sample program follows:

```
>PAS THIRD,TI:=THIRD

THIRD                OMSI Pascal V1.2B RSX  22-Feb-80 22:14 Site #1-1   Page 1
Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     program Third (Output)
   2                     begin
                         ^
******           Expected 'SEMICOLON' missing

   3     1     1     1      Writeln ('"Things get hazy if you stare at them"');
   4     2     1     1   end.

Errors detected:  1
Free memory:  6104  words
Errors detected:  1
Free memory:  6104  words
```

The listing is printed in pages, with a headline on each page showing the program name, the exact version of the Pascal system, the date and time, and the licensed site identification.

Four columns of numbers appear on the left side of each page. The first column, labeled Line, simply 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 number starts at 1 for each control section, and increases by one as each statement is compiled. An up-to-date listing can be useful while debugging, because the statement numbers are used by the Debugger to identify breakpoints.

To illustrate the Level and Nest columns, a more complex program is needed:

```
>PAS FINAL,TI:=FINAL

FINAL                OMSI Pascal V1.2B RSX  22-Feb-80 22:14 Site #1-1   Page 1
Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     program Final (Output);
   2
   3                     const Reality = True;
   4
   5                     procedure Objective;
   6                     begin
   7     1     2     1       if Reality
   8     2     2     2          then Write ('"Things become infinitely complex ')
   9     3     2     2          else Write ('In the Beginning, ... ');
  10     4     2     1     end;
  11
  12                     procedure Awareness;
  13                     var Eye: (Subject, Object);
  14                     begin
  15     1     2     1       for Eye := Subject to Subject do
  16     2     2     2          Writeln ('as one understands them"');
  17     3     2     1     end;
  18
  19                     begin
  20     1     1     1       Objective;
  21     2     1     1       Awareness;
  22     3     1     1     end.

Errors detected:  0
Free memory:  5981  words
>MAC FINAL=FINAL
>TKB FINAL/FP/CP=FINAL,[1,1]PASLIB/LB
>RUN FINAL
"Things become infinitely complex as one understands them"
```

The Level column shows the depth of procedure nesting. The main program is at level 1, its procedures are level 2, and so on; a procedure at level 4 is enclosed by two surrounding procedures or functions. The Nest column shows a similiar nesting of statements within other structured statements.

The PAS command can include several source files, which are combined in sequence to form the complete program. The compilation process can also be modified by the use of "switches" in the PAS command line. A switch is a slash (/) followed by a letter. The most commonly used switches are illustrated in the following examples - see the Programmer's Guide for a complete list.

To demonstrate, let's compile and list the program E. This program calculates an approximation of e (the base of the natural logarithms) by summing the series

$$1+1/1!+1/2!+1/3!+ \ldots +1/N!$$

to the point where additional terms do not affect the approximation:

```
>PAS È,TI:=E

.MAIN.                OMSI Pascal V1.2B RSX  22-Feb-80 22:15 Site #1-1   Page 1
Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     var e, delta, fact: real;
   2                         N: integer;
   3                     begin
   4     1     1     1     e:= 1.0; N:= 1; fact:= 1.0; delta:= 1.0;
   5     5     1     1     repeat
   6     6     1     2       e:= e+delta;
   7     7     1     2       N:= N+1; fact:= fact*N; delta:= 1/fact;
   8    10     1     2     until e = (e+delta);
   9    11     1     1     writeln('With ',N:1,' terms, the value of e is',e:18:15);
  10    12     1     1   end.

Errors detected:  0
Free memory:  6063  words
>MAC E=E
>TKB E/FP/CP=E,[1,1]PASLIB/LB
>RUN E
With 11 terms, the value of e is 2.718280000000000
```

We can watch the progress of the computation and display intermediate values without making any program changes. For this, we use the /D/S switch pair to compile with the interactive Debugger. After compiling, we set a stored breakpoint command to display the current value of e (see the Debugger manual for details of these commands).

Note the third output file in this example, and in the Profiler compilation; this is the symbol table file (.SYM) used by the Debugger and the Profiler.

```
>PAS E,E,E=E/D/S
>MAC E=E
>TKB E/FP/CP=E,[1,1]PASLIB/LB
>RUN E
PASCAL On-line Debugging System -- 24-Apr-79

POD - program name? E
} B(MAIN,6)<W(E);C>
} G
Breakpoint at MAIN,6  e:= e+delta;
1.000000
Breakpoint at MAIN,6  e:= e+delta;
2.000000
Breakpoint at MAIN,6  e:= e+delta;
2.500000
Breakpoint at MAIN,6  e:= e+delta;
2.666667
Breakpoint at MAIN,6  e:= e+delta;
2.708333
Breakpoint at MAIN,6  e:= e+delta;
2.716667
Breakpoint at MAIN,6  e:= e+delta;
2.718056
Breakpoint at MAIN,6  e:= e+delta;
2.718254
Breakpoint at MAIN,6  e:= e+delta;
2.718279
Breakpoint at MAIN,6  e:= e+delta;
2.718282
With 11 terms, the value of e is 2.718280000000000
Program terminated at MAIN,12  end.
} ^Z
```

The computed value is printed with 6 significant digits. For more precision, we can use the /X switch which means "extended precision". With extended precision, 15 significant digits are computed and displayed - see the Programmer's Guide.

```
>PAS E=E/X
>MAC E=E
>TKB E/FP/CP=E,[1,1]PASLIB/LB
>RUN E
With 19 terms, the value of e is 2.718281828459050
```

Finally, let's "profile" the program using the /D/S switch combination, and adding the PROFIL module to the Task Builder input. The leftmost column of the profile listing shows exactly the number of times each line is executed. This allows us to concentrate attention on the parts of the program which might effectively be optimized.

```
>PAS E,E,E=E/S/D
>MAC E=E
>TKB E/FP/CP=E,[1,1]PROFIL,[1,1]PASLIB/LB
>RUN E
Program name? E
Output profile to: TI:
With 11 terms, the value of e is 2.718280000000000

        .MAIN.                OMSI Pascal V1.2B RSX  22-Feb-80 22:16 Site #1-1   Page 1
        Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

        Line  Stmt Level  Nest   Source program

           1                     var e, delta, fact: real;
           2                         N: integer;
           3                   begin
     1     4     1     1     1     e:= 1.0; N:= 1; fact:= 1.0; delta:= 1.0;
     1     5     5     1     1     repeat
    10     6     6     1     2       e:= e+delta;
    10     7     7     1     2       N:= N+1; fact:= fact*N; delta:= 1/fact;
    10     8    10     1     2     until e = (e+delta);
     1     9    11     1     1     writeln('With ',N:1,' terms, the value of e is',e:18:15);
     1    10    12     1     1   end.

        Errors detected:  0
        Free memory:  6055  words
```

Thus ends the guided tour of OMSI Pascal-1. From here, we can suggest several places to find additional knowledge:

(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 -- "you can't hurt the computer!"

(2) Programming in Pascal, by Grogono -- a good course in Standard Pascal, with lots of sample programs for (1), above.

(3) This manual -- for fine points and grubby details of OMSI Pascal-1, it's "the only place in town".

For the serious student, the following books are available from Oregon Software:

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

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

And we recommend joining the Pascal Users' Group, which publishes an excellent newsletter -- send $6 for a one year subscription:

Pascal Users' Group
Attn: Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342
(404) 252-2600

Who is OMSI?

OMSI is not Oregon Minicomputer Software, Inc. -- it's the Oregon Museum of Science and Industry, where we began writing software in the Research Laboratory in the basement. OMSI is a private educational organization whose charter is "to further the education of the youth of the community". Seven of us came from OMSI to found Oregon Software in September, 1977. The name has stuck with us, and we continue to contribute our personal time and corporate resources to the Museum.

But -- we're Oregon Software, please.

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

Director of Research
OMSI
4015 SW Canyon Road
Portland, Oregon 97201
(503) 248-5943

What is OMSI Pascal-2?

Pascal-2 is our new compiler, still under wraps as of this writing. It's an optimizing compiler, written in Pascal -- it's designed to be portable, and has already been moved to a Honeywell computer. The Pascal-2 compiler is bigger and slower than Pascal-1, but not the generated code -- typical programs are 40% smaller and almost twice as fast. You can expect Pascal-2 compilers to be available on a wide range of popular 16 and 32 bit processors in the next several years. Supported users of OMSI Pascal-1 will receive substantial discounts on their purchase of OMSI Pascal-2 licenses for the PDP-11.

OMSI Pascal-1 V1.2 Language Specification

The Language Specification contains details of extensions and limitations of OMSI Pascal-1 as compared to Standard Pascal. Standard Pascal was first defined in the Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth. A further definition is available in the draft proposed Standard from the British Standards Institution (BSI). The draft BSI Standard is being considered for acceptance as an international standard by the International Standards Organization (ISO) and the American National Standards Institute (ANSI). The original Report and the draft BSI Standard are in general agreement. Where the Report and the Standard differ, this document will give a specific reference.

January 3, 1980

Contents

1.0 Syntax Extensions

This section describes extensions to the formal structure of Pascal which are of general utility.

1.1 Program heading

The program heading is optional in OMSI Pascal-1 programs, and it may be omitted entirely. If the program heading appears, the program name will be printed on each page of the program listing. The first six characters of the name will be used as the external name of the object module. Parameters appearing in the program heading are ignored.

1.2 Declaration ordering

The ordering of global declaration sections (CONST, TYPE, VAR, LABEL) is extended in OMSI Pascal-1. Declaration sections may appear more than once and in any order, so long as identifiers are defined before being used.

One application of this is the concatenation of source modules with main programs which provides a primitive source library capability.

Example - compiler input PLOT,MAIN:

```
(* define source module PLOT *)
VAR ...   (* global plotter variables *)
PROCEDURE  (* and plotter functions *)
PROCEDURE ...
(* end of plotter module *)

(* program file MAIN *)
VAR ...   (* global variables *)
BEGIN (* main program code *) END.
```

1.3 Comment brackets

OMSI Pascal-1 provides three forms of comment brackets: the Standard braces {...}, the Standard alternate for upper-case terminals (*...*), and the additional form /*...*/. These may be interchanged freely - it is not necessary for opening and closing comment brackets to have the same form. Comments may not be nested.

Examples:

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

1.4 ELSE clause in CASE statements

OMSI Pascal-1 allows an optional ELSE clause to appear in a CASE statement. It indicates a statement which is to be executed if the CASE selector expression does not match the value of any CASE label. If included, the ELSE clause follows all other statements inside the CASE statement. If no ELSE clause appears and no statement is selected, control passes to the statement following the CASE statement.

Example:

```
repeat
  Readln(Ch);
  case Ch of
    'A','a': Append;
    'D','d': Delete;
    'I','i': Insert;
    'N','n': Newfile;
    'Q','q': ;
    else Writeln('"',Ch,'" is not a legal command');
  end;
until (Ch = 'Q') or (Ch = 'q');
```

1.5 EXIT statement

The EXIT statement terminates the immediately enclosing iterative statement (WHILE, REPEAT, FOR).

The EXIT statement is included for compatibility with previous versions of OMSI Pascal-1. Its use is not recommended in programs intended to be portable.

Example (table search):

```
Found := False;
for I := 1 to Tablesize do
  if Table[I]=Key
    then begin
      Found := True;
      exit;
end;
```

1.6 EXTERNAL Procedures and Functions

The keyword EXTERNAL provides access to separately compiled subroutines and to program libraries and overlay facilities. EXTERNAL appears in the place of a procedure or function body to indicate that the procedure or function is compiled separately.

The compiler will generate references to an external (global) symbol. The first six characters of the procedure or function identifier must form a unique external symbol. References to an external procedure or function are resolved at link or task build time.

Note that the compiler is unable to check parameter types at an external interface.

Examples:

```
procedure Erase; external;
function Rad50(A,B,C: char): Unsigned; external;
```

1.7 FORTRAN Procedures and Functions

The directive 'FORTRAN' is similiar to the EXTERNAL directive. The compiler will generate a calling sequence corresponding to the Digital PDP-11 standard calling sequence, with register 5 (R5) pointing to an argument list. The FORTRAN directive enables calling of external MACRO and FORTRAN subroutines. The FORTRAN calling sequence passes parameters by reference, so the corresponding Pascal parameters must be declared as VAR parameters.

The FORTRAN directive generates the proper call sequence for FORTRAN subroutines, but calling FORTRAN subroutines which perform I/O is operating system dependent. RSX FORTRAN and Pascal share the FCS library without difficulty; the RT11 FORTRAN I/O library requires initialization which is not provided by OMSI Pascal; RSTS/E FORTRAN I/O requires RT11 system calls which are not supported by OMSI Pascal.

Example:

```
function Difference(var X,Y: Real): Real; fortran;
```

2.0 Low-Level Interface

The low-level interface section describes those OMSI Pascal-1 extensions which are useful to programmers who need access to machine dependent PDP-11 characteristics.

2.1 Octal (Base 8) Numbers

Integer constants may be written in octal notation by appending the capital letter 'B' to the number. This applies only to compile-time constants -- runtime integer conversions via Read() are performed using decimal notation.

Example: const TabCode = 11B; (* ASCII tab character *)

2.2 Unsigned Integers

The predefined type Integer has the subrange (-32768 .. 32767) and uses the PDP-11 signed arithmetic operations. Unsigned integers may be specified with the subrange 0..65535. The compiler will generate the unsigned comparison operations of the PDP-11 and will not detect multiplication and division overflow of unsigned integers.

Unsigned integer operations apply only to integer calculations. I/O conversions and conversions to and from Real values are always signed integer operations.

Example: type Unsigned=0..65535;

2.3 Logical operations on Integers

The Boolean operators AND, OR, and NOT are extended to Integer operands. The operators perform the Boolean operations on all 16 bits of their operands. This allows testing or setting of individual bits within a word (for instance, status bits within a device register).

Example: Byte := Ord(Ch) and 377B;

2.4 References to fixed (absolute) memory

OMSI Pascal-1 allows the keyword ORIGIN to appear in variable declarations, associating a variable identifier with a specific memory address. This provides access to fixed memory addresses,

such as device control registers or operating system parameter blocks.

Example (read directly from the RT11 console):

```
const Ready=200B;
var   KbCsr origin 177560B, KbBuff origin 177562B: Integer;
      Ch: Char;
begin
      while (KbCsr and Ready)=0 do (* nothing *);
      Ch := Chr(KbBuff);           (* get character *)
end;
```

2.5 Address operator (@)

OMSI Pascal-1 provides a unary address operator, indicated by the @ character. When applied to a variable of type T, it yields a value of type ^T (pointer to T). The address operator can be used to link variables into list structures or (more commonly) to pass variable addresses to low-level routines.

Example:

```
var  Buffer: Block; XRLoc origin 446B: ^Block;
begin
    XRLoc:= @Buffer;  (* pass address to RSTS/E *)
end
```

2.6 Embedded assembly code

PDP-11 MACRO assembly code may appear at any point in an OMSI Pascal-1 program. Assembly code sections have the form of a Pascal comment, beginning with the $C embedded switch. Any MACRO-11 feature may be used within embedded code. The compiler provides some assistance in accessing Pascal variables, though the programmer is expected to have some understanding of the OMSI Pascal-1 runtime environment. Note that the default radix within a Pascal-produced MACRO file is decimal, not octal.

Example:

```
procedure EmtTrap(N:Integer);
begin
  (*$C
    MOV N(SP),-(SP) ; push parameter N
    EMT 53          ; call EMT handler
  *)
end (*EmtTrap*);
```

3.0 I/O Support Extensions

I/O support extensions provide the OMSI Pascal-1 programmer with additional control of the interface to the operating system.

3.1 Reset()/Rewrite() optional parameters

Three additional parameters may appear following the file variable in calls to the Reset() and Rewrite() standard procedures. These optional parameters allow the program to dynamically bind a file variable to an external file and provide status and error information.

The general form is:

```
Reset( F , Filename , DefaultName , Size )
```

where the parameters have these types:

```
F - any file variable
Filename - literal string, or (packed) array of Char
DefaultName - same as Filename
Size - Integer variable
```

Reset(F,Filename) connects the file variable F with the external file identified by Filename. Filename conforms to the operating system conventions, and may contain device, filename, extension, and other fields such as PPN/UIC and version number. The Filename parameter may also contain switches specifying access modes or other special characteristics. If the external file does not exist prior to the Reset(), a fatal error will result. Upon successful completion of a Reset(), either the file buffer F^ will contain the first element of the file, or Eof(F) will be True.

Reset(F,Filename,DefaultName) performs the same function, with DefaultName having the same format as Filename. Fields of the external name which are not specified in Filename are filled from the information in DefaultName. Common default fields are the extension, protection code, and mode switches.

Reset(F,Filename,DefaultName,Size) provides a recovery capability on file open errors. Size must be a variable (VAR parameter). After a successful Reset(), Size contains the length of the file in blocks. If an error occurs, Size is set to negative one (-1).

```
Rewrite( F , Filename , DefaultName , Size )
```

Rewrite() creates a new external file. The optional parameters have the same meaning as in Reset() with one addition: Size specifies the initial storage, in blocks, to be allocated for the file.

Reset() and Rewrite() may be applied to the standard files Input and Output respectively. This will redirect the default input or output streams to the specified file instead of the user terminal. A subsequent Close() will break the connection and reconnect the default file to the terminal.

Example:

```
program Copy; (* copy to printer *)
var Name: array[1..20] of Char;
    Ch: Char; Len: Integer;
begin
  repeat                    (* Get a Filename and Reset() it *)
    Write('File: ');
    Readln(Name);
    Reset(Input,Name,'.PAS',len)
  until Len <> -1;          (* until not error code *)
  Rewrite(Output,'LP:');    (* redirect Output to printer *)

  while not Eof do begin    (* copy Input to Output *)
    while not Eoln do begin
      Read(Ch); Write(Ch);
    end;
    Readln; Writeln;
  end;
end.
```

3.2 Seek() procedure

The predefined procedure Seek() causes direct positioning of a file window variable to any desired component of the file.

```
Seek( F , Index )
```

F may be of any file type except Text, and must be connected to an external file which supports direct access (typically disk or DECtape). Index is an unsigned integer expression which specifies the desired component. File components are numbered sequentially beginning with one (1). If Index specifies a number greater than the number of components actually present, then Eof(F) is set to True.

To read component N of file F, use:

```
Seek(F,N); (* component N is available in F^ *)
```

To write component N, use the sequence:

```
Seek(F,N); (* position to component N *)
F^ := ();  (* assign new value *)
Put(F);    (* write component to file *)
```

If the Put() in the above sequence is omitted, the effects will be unpredictable and the new data may be lost.

Sequential I/O operations such as Get() and Put() may be mixed with Seek() and will advance the file window to the next component. Reset(F) is equivalent to Seek(F,1).

The direct access extension bypasses the Standard Pascal restriction prohibiting simultaneous read and write access to a file. For this reason, direct access files are identified by the '/Seek' switch which must appear in the Filename or DefaultName field of the associated Reset() or Rewrite().

3.3 Break() procedure

For efficiency, OMSI Pascal-1 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.

3.4 Close() procedure

Close(F) indicates that the program has completed processing the file F, and that internal buffer storage may be reclaimed. Close(F) removes any connection to an external file, so that Reset(F) or Rewrite(F) must precede any subsequent operations with that file variable.

3.5 Readln() Array of Char

Read() and Readln() will read characters from a Text file into a (packed) array of characters. Reading begins at the current file position and continues until either the array is filled, or Eoln() is True, in which case the remainder of the array is filled with blanks.

3.6 Write() Array of Char

In accordance with the draft proposed ISO Standard, a Write() 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.

Example:

```
Write(Buffer:BuffCount);  (* write buffered characters *)
```

3.7 Write() Octal (Base 8)

Write() will write integers in octal notation if the field width specification is negative.

Example: Write(I:-5); (* Display octal value of I *)

3.8 Interactive 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 can present serious difficulties when applied to files which 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 type the first input character (or line) prior to the execution of the first program statement.

OMSI Pascal-1 takes the following 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 solves most of the problems with interactive terminals in a predictable manner, but one should note that this approach creates other difficulties. 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: array[1..72] of Char;
    Count: Integer;
begin
  Count := 0;
  while not Eoln do begin
    Count := Count+1;
    Read(Line[Count]);
  end;
  Readln;
end;
```

4.0 Additional Predefined Functions

OMSI Pascal-1 provides some additional built-in functions.

4.1 Time function

The Time function takes no parameters and returns a real value which corresponds to the current time of day. The Time is represented in hours after midnight, so that 9:30 AM is 9.50 and 1:45 PM is 13.75. The exact resolution of the Time function 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: 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;
```

4.2 Exp10() and Log() functions

The Exp10() and Log() functions are similiar to the standard Exp() and Ln() functions, but with a logarithm base of ten (10).

5.0 Non-Standard Language Elements

This section describes the elements of OMSI Pascal-1 which do not conform to the accepted definition of Standard Pascal.

5.1 Pack() and Unpack() not available

The reserved word PACKED may appear in type definitions, but it has no meaning in OMSI Pascal-1 programs. Packed types require the same amount of storage as unpacked types. The standard procedures Pack() and Unpack() are not available. The following equivalent FOR statements can be used instead:

```
var A: array[M..N] of T;
    Z: packed array[P..Q] of T;
  for J:= P to Q do Z[J]:= A[J-P+I]; { Pack(A,I,Z) }
  for J:= P to Q do A[J-P+I]:= Z[J]; { Unpack(Z,A,I) }
```

5.2 Program Parameters

Program parameters (identifiers appearing in the program heading) have no meaning in OMSI Pascal-1 programs. The program heading may be omitted entirely if desired. External files can be declared by using the Reset() and Rewrite() procedures with optional parameters.

5.3 Identifier Scope Rules

In Standard Pascal, the scope of an identifier (that section of the program within which the identifier indicates a particular object) is directly related to the block structure. A definition of an identifier in a procedure, for example, prohibits that identifier from indicating another object throughout the entire procedure.

OMSI Pascal-1 uses a subtly different rule for the scope of an identifier, called 'one-pass' scope, in which a definition of an identifier prohibits only subsequent uses of the identifier within the block from indicating an object outside the block.

The non-standard scope rule is described here for completeness, but it is of little concern to the programmer. Indeed, the majority of Pascal compilers use the identical (incorrect) rule.

5.4 Read()/Write() Text files only

In the 1978 printing of the Pascal User Manual and Report, the Read() and Write() standard procedures were extended to apply to all file types. This extension has not yet been incorporated into OMSI Pascal-1, so that Read() and Write() are applicable only to files of the standard type Text.

The following substitions may be used:

```
For Read(F,V), use:  V:=F^; Get(F);

For Write(F,V), use: F^:=V; Put(F);
```

5.5 Eof() not accurate (RT11, RSTS only)

On the RT11 and RSTS operating systems, a file is structured as a sequence of 512 byte blocks. No finer resolution is available as to the end of data in the last block. Therefore, the Eof() standard function can not be relied upon as accurate, and another method (sentinel record, record count) should be used to indicate the end of usable data.

Note that this problem does not apply to Text files, where Eof() is identified correctly.

6.0 Implementation Definitions

This section provides specific details and characteristics of implementation-defined elements of OMSI Pascal-1.

6.1 Identifiers

OMSI Pascal-1 permits identifiers to be of any length, and all characters are significant. Lower case letters may be used and are interpreted the same as upper case, so that "name", "Name", and "NAME" are equivalent identifiers.

Due to limitations of the object program file structures, the first six characters of any EXTERNAL or FORTRAN identifier must form a unique external name.

6.2 Standard type Integer

The standard type Integer has the range (-32768..32767). Unsigned integers may be declared using the subrange notation 0..65535. Note that arithmetic overflow is detected only for multiplication and division of signed integers.

The predefined identifier Maxint has the value 32767.

6.3 Standard type Real

Real variables have the standard PDP-11 single or double precision floating point structure, with the range 1E-38 .. 1E+38. Single precision values give 7 decimal digit precision; extended (double precision) values give 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 15 decimal digits in extended precision.

6.4 Standard type Char

OMSI Pascal-1 uses the 7-bit full ASCII character set. Characters are stored as signed bytes with all 8 bits available to the programmer, so that Ord(Char) has the subrange (-128..127).

6.5 Standard type Text

The standard type Text is a file type with components of type Char, with RSTS/E and RT11 input masked to the 7-bit ASCII set minus the NUL (0) character. On RSX systems, the standard function Eoln() is set by the end of a file record; on RSTS/E and RT11 systems by the LF (10) or ESC (27) character codes.

The standard procedures Read(), Readln(), Write(), Writeln(), and the standard function Eoln() are applicable only to Text files. The Seek() procedure is not recommended for use with Text files.

6.6 SET types

OMSI Pascal-1 limits sets to a maximum of 64 elements. The 64 element maximum forms a subrange which is not required to have a lower bound of zero, but may instead be positioned at any 64 element (or smaller) subrange of a base type (for example: 100..150, -25..25).

A set of the standard type Char is equivalent to the set of Chr(32)..Chr(95), which is a subset of ASCII containing the upper case letters, digits, punctuation symbols, and the space character, but lacking the control characters and lower case letters.

6.7 New() and Dispose() procedures

In allocating storage for variant records, the New() procedure will allocate memory for the largest variant; any tag field values specified to New() and Dispose() are ignored.

Storage must be explicitly released with Dispose() -- no automatic garbage collection is performed. Storage occupied by variables passed to Dispose() is reclaimed for use by the New() procedure. Dangling pointer references are not detected.

6.8 Procedural Parameters

The passing of PROCEDURE and FUNCTION parameters is supported by OMSI Pascal-1 with the syntax described in the Pascal User Manual and Report (the proposed ISO Standard differs in this area).

Predefined procedures and functions are not permitted as procedural parameters. This can be bypassed by declaring a second procedure which calls the standard procedure, and which can itself be used as a procedural parameter.

Example:

```
function Sine(X: Real): Real;
begin
  Sine:= Sin(X)
end;
```

6.9 Implementation Limitations

The PDP-11 has six general purpose registers. In OMSI Pascal-1, one register (R5) is always allocated for access to global variables, and another (R4) is allocated in some blocks for access to intermediate level variables. The remaining registers are used for integer calculations, address computations, and WITH statement variable access. Each WITH statement uses one register for the duration of the enclosed statement. This implies a maximum nesting of WITH statments of three levels. Complex expression calculations can also exceed the available registers. If the 'Out of registers' error occurs, remove a WITH statement or simplify the indicated expression by calculating intermediate results.

The syntactic nesting of procedures is limited to a depth of 10 levels. There is no implementation restriction on the actual depth of recursion of a program, although unlimited recursion will eventually cause the program to exceed available memory.

6.10 Error Detection

OMSI Pascal-1 does not detect the following runtime errors:

Uninitialized variables
Subrange bounds exceeded
Integer overflow
Real underflow
Record variant mismatch
Dereference of NIL pointer

The following runtime errors are detected:

Stack overflow
Heap overflow
Real overflow
Integer multiply/divide overflow
Array bounds exceeded
Dispose() of NIL or duplicate pointer
Incorrect numeric format
I/O errors

Predefined Identifiers

Constants		
	False,True	
	Maxint	
Types		
	Boolean	
	Char	
	Integer	
	Real	
	Text	
Variables		
	Input, Output	
Functions		
	Abs	
	Arctan	
	Chr	
	Cos	
	Eof	
	Eoln	
	Exp	
	Exp10	Base 10 Exponential
	Ln	
	Log	Base 10 Logarithm
	Odd	
	Ord	
	Pred	
	Round	
	Sin	
	Sqr	
	Sqrt	
	Succ	
	Trunc	
	Time	Time of day
Procedures		
	Break	Transmit buffered output
	Close	Close file
	Dispose	
	Get	
	New	
	Page	
	Put	
	Reset	
	Rewrite	
	Read	
	Readln	
	Seek	Direct access I/O
	Write	
	Writeln	

Reserved Words
(* extensions)

And
Array
Begin
Case
Const
Div
Do
Downto
Else
End
* Exit
* External
File
For
* Fortran
Forward
Function
Goto
If
In
Label
Mod
Nil
Not
Of
Or
* Origin
Packed
Procedure
Program
Record
Repeat
Set
Then
To
Type
Until
Var
While
With

Compilation Switch Options

The compilation process and the resulting program can be modified by switches appearing in the PAS command. Switches are a single alphabetic character following the '/' (slash) marker, as in the command PAS PROG=PROG/X.

The complete set of compilation switches appears below, followed by a detailed description of each switch.

/D	Debug	Debugger compilation
/E	External	External module compilation
/F	Fast reals	Generate calls rather than traps
/L	Listing	Produce compilation listing
/N	Nolist	List errors only
/S	Source	Include source lines (modifies /D)
/X	eXtend	Extended precision Reals (15 digits)

Listing Control Switches (/L, /N)

The /L switch overides embedded listing switches, and directs the compiler to produce a listing. The /N switch directs the compiler to list only lines in error. The /L and /N switches are related to the $L+ and $L- embedded switches.

Real Arithmetic Switches (/X, /F)

The /X switch causes the compiler to use extended precision for values of type Real. All Real values are extended -- it is not possible to mix normal and extended precision values. The /X switch is related to the $X embedded switch. See the section on Extended Precision.

The /F switch is of limited utility. On processors lacking both FPP and FIS floating point hardware, Real operations are normally performed by trapping each FIS instruction and simulating its effects. The trapping process requires some overhead, but is compact. The /F switch causes the compiler to generate subroutine calls instead, which are faster but require an extra word for each floating point operation.

Debugger Switch (/D)

The /D switch indicates a Debugger or Profiler compilation. This switch requires the specification of a symbol table file and, if /S is also present, a listing file. The /D switch causes generation of code to identify each procedure and statement to the interactive Debugger or Profiler. The /D switch is related to the $D+ and $D- embedded switches. See the section on the Debugger.

Source Mode Switch (/S)

The /S switch performs two distinct functions. When used with the Debugger switch (/D/S), it enables the source program mode of operation and connects the actions of the Profiler and Debugger to the source text of the program.

The /S also causes the assembler output file to include the Pascal source lines embedded as comments within the assembly file. This use of the /S switch is related to the $S+ and $S- embedded switches.

External Module Switch (/E)

The /E switch indicates an external module compilation. This causes the outermost procedures and functions to be identified to the Linker with global entry names. An external module can include global declarations, procedures, and functions but is not required to include a main control section. The /E switch is related to the $E embedded switch. See the External Module section.

The Task Builder

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

The basic Task Builder command is:

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

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

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

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

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

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

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

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

There are two options which are commonly used with Pascal programs. The UNITS option increases the number of Logical Unit Numbers (LUNs) which are available to the program. The number of LUNs available determines the maximum number of files which may be open at any time. Two LUNs (5 and 6) are always used by Pascal for the standard files Input and Output; if the Debugger is in use, it requires two LUNs for its operation. There are 6 LUNs allocated by default, so a program using three or more files should allocate more LUNs with the UNITS option as shown below:

```
>TKB
TKB>MAIN/FP/CP=MAIN,[1,1]PASLIB/LB
TKB>/
Enter Options:
TKB>UNITS=9
TKB>//
```

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

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

The full capabilities of the Task Builder are described in the Task Builder Reference Manual (DEC-11-OMTBA-C-D). See also the Overlays section of this manual.

I/O Control Switches

The Reset() and Rewrite() standard procedures accept additional arguments specifying a Filename of an external file, and a DefaultName with default fields of the filename. 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 Filename or DefaultName parameters as in this example:

```
Rewrite(F,'data/si:12','.dat/seek/span');
```

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

/BLK (Blocked) -- records in the file are not to cross disk block boundaries. This allows faster access at the cost of additional space. This switch is the default for record files.

/SPAN
/NOBLK (Spanned) -- records in the file are allowed to cross disk block boundaries, making most effective use of space. This is the default for variables of type Text.

/ALOC:n
/CL:n (Allocation or Clustersize) -- the parameter N determines the allocation unit for each extension of the file. A positive value for N indicates a contiguous allocation; a negative value indicates a non-contiguous allocation.

/SI:n (Size) -- used with Rewrite() to specify the initial allocation of space for the file. A positive value for N will allocate contiguous blocks; a negative value will allocate non-contiguous blocks.

/CR (Carriage control) -- when printed, records in the file are to be preceded by a LF character, and terminated with a CR character. This is the default for Text variables.

/NOCR (No carriage control) -- the default for non-Text file variables.

/FTN (FORTRAN carriage control) -- the first character of each record determines the line spacing before displaying the record, as per the FORTRAN conventions.

/RO (Read-only) -- only read accesses to the file are permitted. This is the default for files initially opened with Reset().

/RW (Read-write) -- both read and write access permissions are available. This is the default for files initially opened with Rewrite().

/SEEK (Direct-access) -- permits use of the Seek() standard procedure, and allows both Get() and Put() operations on the file variable. The /RW switch should be used in combination with /SEEK for update access to files opened with Reset().

/TEMP (Temporary) -- marks the file for deletion upon Close() or program termination. A file created with no file name, as in Rewrite(F), will also be marked as a temporary file.

/NSP (No supersede) -- when creating a file with Rewrite(), this switch will cause an error if a file of the same name already exists.

/SHR (Share) -- permits shared access by multiple users to the file. Note that OMSI Pascal-1 offers no built-in facilities for record locking.

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

/ACTL:n Sets F.ACTL to the parameter N. F.ACTL determines the number of retrieval pointers and magtape positioning characteristics.

/APD Sets FA.APD in F.FACC; this indicates that records are to be appended to an existing file.

/EXT Sets FA.EXT in F.FACC; allows extension of the file.

/INS Sets FD.INS in F.RACC; indicates that Put() operations in sequential mode should update the record and not truncate the file.

/WRT Sets FA.WRT in F.FACC, which provides write access to the file.

/FIX:n Sets R.FIX in F.RTYP. This indicates a file of fixed length records of length N; this is the default file type for non-Text files.

/VAR:n Sets R.VAR in F.RTYP to indicate a file of variable length records, with a maximum length of N bytes. The default type for Text files is /VAR:132.

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

The Profiler

The Profiler is a program measurement tool which can be used to identify the sections of a program that can be most effectively optimized. Empirical measurements show that typical programs consume a large fraction of their computation time in a small portion of the program code ("90% of the time in 10% of the code"). The Profiler counts the actual number of times each statement is executed and each procedure is activated, and displays this information either in the program listing or in a tabular form.

The /D switch causes compilation for the Profiler or the Debugger, which use the same interface. The PROFIL module is included at Task Build time. The /S switch is recommended in addition for more convenient display of the profile information.

When the Profiler begins execution, it will ask for the program name. The Profiler uses the symbol table and listing files produced by the compiler to identify procedures and statements in the program. The symbol table file normally has the same name as the program and the extension .SYM, and the listing file normally has the extension .LST. The Profiler will ask for the correct filenames if the normal files are not available.

The Profiler will then ask for the desired destination of the profile information. The profile will be written to the specified file with the default extension .PRO. This should be a permanent file (disk or hard copy device), as the Profiler requires roughly a factor of fifty performance overhead while gathering information.

The program being measured will then execute normally, although somewhat more slowly. Upon normal termination, or any fatal error, or ctrl/C interrupt, the profile information will be written to the specified file.

The first section of the profile is the Procedure Reference Profile, which lists each referenced procedure and function with the count of calls on that procedure. The second section is the Statement Reference Profile, displayed in tabular format. If the /S (source) switch is specified, this section displays the program listing with an additional column containing the reference count for each line.

The Profiler is limited in several respects: only the first 100 statements in each procedure will be counted, and a maximum of 40 procedures and functions can be profiled. The $D- and $D+ embedded switches can be used to selectively enable and disable profiling.

Example:

```
PRIMES                OMSI Pascal V1.2B RSX  22-Feb-80 22:52 Site #1-1   Page 1
Oregon Software  2340 SW Canyon Road  Portland, Oregon 97201  (503) 226-7760

Line  Stmt Level  Nest   Source program

   1                     program Primes;        (* Author: N. Wirth *)
   2                     const  N=2500; (* first 2500 Primes *)
   3                     type   Index=1..N;
   4                     var    X,Square: integer;
   5                            I,K,Lim: Index;
   6                            Prime: Boolean;
   7                            P: array[Index] of integer;
   8                            V: array[1..100] of integer;
   9                     begin
   1    10     1     1     1      P[1]:=2; write(2); X:=1; Lim:=1; Square:=4;
   1    11     6     1     1      for I:=2 to N do begin
2499    12     8     1     3        repeat
11153   13     9     1     4          X:=X+2;
11153   14    10     1     4          if Square<=X then begin
  35    15    12     1     6            V[Lim]:=Square;
  35    16    13     1     6            Lim:=Lim+1; Square:=P[Lim]*P[Lim];
  35    17    15     1     6            end;
11153   18    16     1     4          K:=2; Prime:=true;
11153   19    18     1     4          while Prime and (K<Lim) do begin
94012   20    20     1     6            if V[K]<X
28669   21    21     1     7              then V[K]:=V[K]+P[K];
94012   22    22     1     6            Prime:=(X<>V[K]); K:=K+1;
94012   23    24     1     6            end;
11153   24    25     1     4        until Prime;
2499    25    26     1     3        P[I]:=X; write(X);
2499    26    28     1     3      end;
   1    27    29     1     1   end.

Errors detected:  0
Free memory:  5967  words
```

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 /X compilation switch or the $X embedded switch cause all Real values to have extended precision. Extended precision values each occupy 4 words of storage, and provide 15 digit precision in all real calculations, including the transcendental functions.

Extended precision applies to all Real values in a program -- it is not possible to mix normal and extended precision variables. All external modules must be compiled with the same precision as the main program, even if no Real variables are present.

Embedded Switches

Embedded switches provide control of compilation options within the Pascal source program. Embedded switches have the form of a Pascal comment beginning with a dollar sign ($), followed by a single uppercase alphabetic character and possibly a plus or minus sign, as in (*$L+*). Several of the embedded switch functions can also be provided by compilation switches. Embedded switches have the advantage that once included in a program, they cannot be accidently omitted from a compilation.

The complete list of embedded switches below is followed by a more detailed description of each switch function. The switches which have +/- signs are counting switches; that is, each occurrence either increments or decrements the switch value, and a positive value enables the switch function. Switches which are initially enabled are marked with [+]; switches marked [MBF] must be first -- they must appear before any Pascal code.

$A-,$A+	Array check	Include array subscript check [+]
$C	Code insert	See the Embedded Assembly Code section
$D-,$D+	Debugger	Include debugger interface
$E-,$E+	External	External module compilation
$F-,$F+	Fast FPP	Enable floating point calls
$L-,$L+	Listing	Source lines in listing [+]
$S-,$S+	Source mode	Source lines in assembly
$T-,$T+	sTack check	Include stack overflow check [+]
$X	eXtend	Extended precision reals [MBF]

Error Checking Switches ($A, $T)

The $A switch controls the generation of code to check array references and ensure that the index is within the subscript range of the array. subscript checking is initially enabled; the $A- switch will disable checking. If enabled, each subscript check requires 8 words.

The $T switch controls stack overflow checking, and is initially enabled. Stack overflow is possible upon entry to any procedure or function block. This switch can be disabled with $T-, resulting in small savings of memory (2 words per procedure).

Debugger/Profiler Switch ($D)

The $D switch controls the interface code to the Debugger and Profiler. If enabled, each statement and procedure includes instructions to call the Debugger or Profiler. These instructions require 1-3 words per statement (1 word for statements 1-255 of each procedure, 2 words otherwise, and an additional word if /S source mode is enabled). In program sections known to be correct,

the Debugger interface may be disabled by preceding the section with {$D-} and concluding the section with {$D+}. The Debugger interface must be enabled at the start of the main program block.

External Module Switch ($E)

Enabling the $E switch causes global procedures and functions to be labeled as external entry points in the relocatable object file. A main program section encountered when the $E switch is enabled is ignored. See the External Module section.

Real Arithmetic Mode Switches ($X, $F)

The $X switch enables extended precision (15 digit) real arithmetic. If present, the $X switch must precede any Pascal code. Note that it is not possible to mix normal and extended precision in one program, so that each module in separate compilations must be compiled with the same precision. See the Extended Precision section.

The $F switch is useful only on processors which lack FIS and FPP hardware for floating point calculations. On these processors, floating point instructions are normally trapped and simulated. The $F switch instead causes direct subroutine calls to floating point routines, saving about 0.2 milliseconds per floating point instruction at the cost of an extra word.

Listing Control Switch ($L)

The $L switch controls the appearance of lines in the program listing file. If enabled, all program text will appear in the listing. If the $L switch is disabled, only lines in error and error messages will appear.

Source Mode Switch ($S)

Enabling the $S switch causes the Pascal source lines to appear in the compiler assembly output as comments. This makes it easier to determine the code generated for each statement.

Format and Cross-Reference (FORMAT)

The FORMAT utility supplied with OMSI Pascal-1 will automatically reformat a Pascal source program, adjusting indentation and partitioning statements so that a program listing reflects the program structure. The FORMAT program can also provide a cross-reference index of a Pascal source program showing block calls, nesting, and identifier references.

The FORMAT command line can contain one or two output files, an input source file, and several optional switches. Run FORMAT as follows:

```
>RUN FORMAT
FORMAT V2.0 (10Dec79)
*<Formatted.PAS>,<Crossref.CRF>=<Source.PAS>/switches
```

The Format output file is the formatted source program. Several switches select token translation options:

/L	Lowercase	Lowercase identifiers, uppercase keywords
/M	Mixedcase	Unchanged identifiers, uppercase keywords
/U	Uppercase	All letters uppercase

The Crossref output file (if specified) normally contains the program listing with line and page numbers, followed by the procedure call and nesting index. Two switch options apply to the cross reference:

/C	Crossref all	Cross reference all identifiers
/N	No listing	Produce only crossref index

The /C switch may be used only for source programs of moderate size, due to memory limitations.

The Improver (IMP)

The utility program IMP decreases the size of the object code produced by OMSI Pascal-1 by replacing branch/jump combinations with single branches when possible. IMP will reduce the generated code by roughly 5 to 8 percent.

IMP asks for the assembler file (INPUT) and the destination of the improved assembler file (OUTPUT). These are usually the same filename.

Because IMP runs quite slowly, it is recommended for use only on completely debugged production programs.

Dynamic String Package

A package of procedures and functions for dynamic string processing is supplied with OMSI Pascal-1 V1.2 in the file STRING.PAS. Written in Standard Pascal, the package supports programs using strings on any Pascal implementation. Strings are stored as a record structure with a fixed maximum number of characters (normally 100 but easily changeable), and an integer marking the current length of the string.

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

Len(S) - returns the current length of string S;

Clear(S) - initializes string S to empty;

ReadString(F,S) - reads a value for string S from the text file F. The string is terminated by Eoln(F) and a Readln(F) is performed. String overflow results in truncation.

WriteString(F,S) - writes the string S to the text file F. The same effect can be achieved by passing the parameter S.Ch:S.Len to Write(), as in Write(F,'S=',S.Ch:S.Len).

Concatenate(T,S) - appends string S to the target string T. The resulting value is string T. Overflow results in truncation.

Search(S,T,Start) - searchs string T for the first occurrence of string S to the right of position Start (characters are numbered beginning with one). The function Search() returns the position of the first character in the matching substring, or the value zero if the string S does not appear.

Insert(T,S,Start) - inserts the string S into the target string T at position Start. Characters are shifted to the right as necessary. Overflow produces a truncated target string. A Start position which would produce a string which is not contiguous 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, and if negative, the substring is to the left.

Delete(S,Start,Span) - deletes the substring defined by Start, Span from the string S.

Substring(T,S,Start,Span) - the substring of string S defined by Start, Span is assigned to the target string T.

External Modules

External modules allow several program sections, each containing at least one procedure, function, or main program, to be compiled independently and combined at Task Build time. External modules may be combined into libraries to simplify handling of common routines. The external module interface also allows inclusion of modules written in other languages, such as FORTRAN and MACRO.

The EXTERNAL directive is used to reference a procedure or function in an external module. The declaration of an external procedure or function contains the procedure or function name and parameters, followed by the directive EXTERNAL (similiar to FORWARD). The procedure or function body does not appear in the program unit referencing the external routine.

The FORTRAN directive replaces EXTERNAL to reference external routines written in FORTRAN or MACRO. The FORTRAN 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 R5 to a list of parameters).

The /E compilation switch and the $E embedded switch are used to create modules which can be referenced by EXTERNAL directives. When the $E switch is enabled, each global procedure and function declaration causes an external (global) symbol to be defined. These global symbols are matched at Task Build time to the global references created by the EXTERNAL directive.

The external reference symbols are composed of the first six characters of the external procedure or function identifier, and must uniquely identify the external routine. Duplication or overlap of external symbols results in the Task Builder error 'Module multiply defines symbol', while a missing module results in the 'Undefined symbols' error message.

One caution should be observed when using EXTERNAL and FORTRAN directives. Parameters to external modules cannot be checked by the compiler for type conformance, so an accidental type mismatch may cause entirely unpredictable results.

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. Again, the compiler cannot verify the conformance of global data.

When combining modules to form libraries, remember that the procedures and functions from one compilation form a single module, and cannot be individually selected from the module. The module name is taken from the first six characters of the program identifier (in the program heading).

Overlays

The Task Builder has the capability of creating overlaid tasks, wherein program sections which are not in use can be overwritten by other sections. The full overlay capabilities are described in chapters 5 and 6 of the Task Builder Reference Manual; an overview is presented here, oriented toward Pascal tasks.

The overlay structure of a program can be very complicated, so there is a special "language" to define overlays called the Overlay Description Language (ODL). An overlay structure is defined in an ODL file (with the extension .ODL), which describes each program section and its position in an overlay tree.

Supplied with OMSI Pascal-1 is an overlay description file ([1,1]PAS.ODL) which contains overlay descriptions for the Pascal runtime library and the FCS system I/O routines. The following overlay structures are defined in PAS.ODL:

SYSIO - a co-tree for the FCS I/O routines used by Pascal

SINGLE - the Pascal library routines for single precision real arithmetic and transcendental operations

DOUBLE - the extended precision library routines

DEBUG, DEBUG1, DEBUG2 - three co-trees that describe the full overlay structure of the Pascal On-line Debugger

Any of these descriptors can be used in an ODL file by giving an indirect file reference to [1,1]PAS.ODL. The example ODL file below builds a task with a main program (MAIN) and an overlaid Debugger.

```
@[1,1]PAS.ODL
.ROOT DEBUG,DEBUG1,DEBUG2,SYSIO,*MAIN
.END
```

(the '*' is necessary -- it causes autoloading of MAIN)

If this ODL file is named DEBUG.ODL, the task can be built with the following Task Builder command:

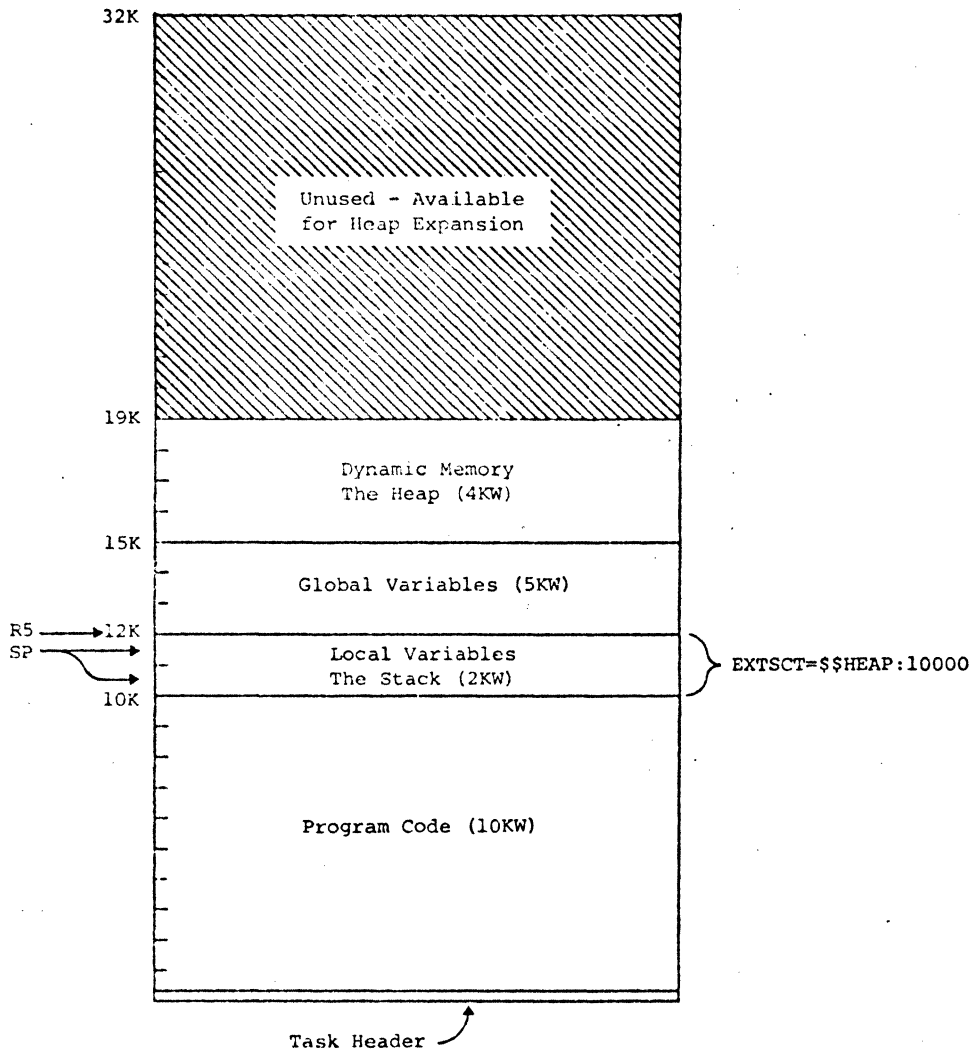
```
>TKB MAIN=DEBUG.ODL/MP
```

The following ODL file builds a task containing a main program (MAIN), and three external modules (SUB1, SUB2, SUB3) which are called only by MAIN and can be overlaid against each other.

```
@[1,1]PAS.ODL
.ROOT SYSIO,SINGLE,*MAIN-*(SUB1,SUB2,SUB3)
.END
```

Runtime Memory Organization

A PDP-11 program has a virtual address space of 32,768 words, or 32KW (1KW is 1024 words). The figure below shows this address space as it might be allocated for a typical program of moderate size.



This figure represents a snapshot taken during program execution, illustrating the division of available memory. Each section is described in the following paragraphs.

Task Header

The Task Header contains task parameters and data required by the Executive and provides a storage area for saving the task context.

Program Code

The Program Code section contains the instructions of the user program, including overlays, external modules, and routines from the runtime library. The size of this division depends entirely on the user program and its overlay structure.

Local Variables - The Stack

The Stack contains all variables local to inner blocks of the program, and is also used for temporary calculations, parameter passing, and subroutine return information. At the time a block is entered, a stack frame is created which contains all information local to that block. Stack frames are created and released in a purely nested fashion. See below for a detailed description of a stack frame.

Memory is allocated for the Stack immediately adajacent to the Program Code section. The Stack is the only division whose allocation can be directly controlled by the user. The size of the Stack is set at Task Build time; the option format is EXTSCT=$$HEAP:nnn, where nnn is the number of bytes (in octal) to be allocated to the Stack. The default size of the Stack is 2K words, or 10000 (octal) bytes.

The current Stack frame is always pointed to by the Stack Pointer (SP, register R6), which points initially to the top of the Stack. As nested Stack frames are allocated, the Stack Pointer decreases in value (points to lower addresses). If the Stack is too small, the Stack Pointer will eventually overrun the Program Code division and cause the 'Not enough memory' error.

Global Variables

The Global Variables section contains the program's global variables - those defined in the outermost, or main, block of the program. The size of this division does not change during program execution.

Register 5 (R5) points to the base of the Global Variables and is used for access to global variables.

Dynamic Memory - The Heap

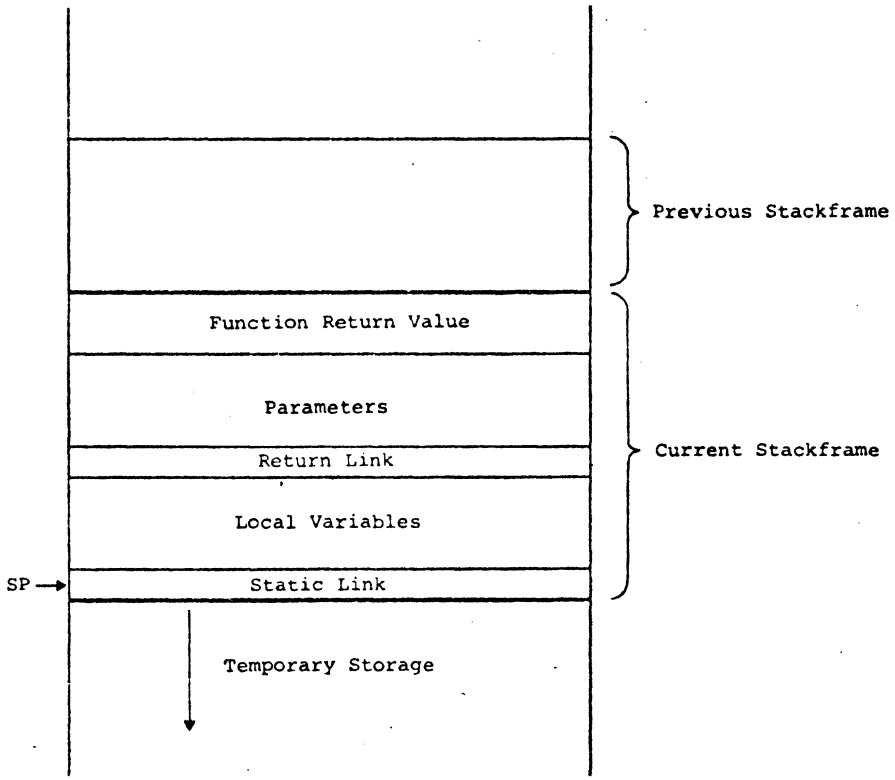
The Heap contains I/O control blocks and buffers, and variables allocated by the New() procedure. The Heap is unique in that it is not allocated any memory initially, but instead expands as necessary. The Heap is allocated adjacent to the Global Variables, and may grow on demand to the upper limit of 32KW or the system maximum set with the /MAXEXT option. When that limit

is reached, any unused memory in the Stack will be used as well. The error message 'Not enough memory' indicates total exhaustion of memory resources.

Dynamic task expansion requires that the Extend Task (EXTK$) directive be included during RSX system generation, and that the task be checkpointable (/CP Task Builder switch). If the task cannot be dynamically expanded, then the EXTSCT=$$HEAP:nnn option must be specified at Task Build time to allocate space for the Stack, Heap, and Global Variables sections.

A Closeup of the Stack - The Stack Frame

The Stack is composed entirely of Stack Frames. A Stack Frame is created during entry to every block (excluding the main program block), and is released when the block is exited. The following diagram illustrates the possible components of a Stack Frame. Most of these components are optional -- only the Return Link is required in every Stack Frame.



Function Return Value

This field is present in Stack Frames associated with function blocks, and holds the value to be returned by the function. Its position at the bottom of the Stack Frame allows it to be 'popped'

from the stack when control returns to the caller of this block.

Parameters

The Parameters field contains either parameter values or their addresses. A block without parameters does not have this field in its Stack Frame.

Return Link

This field is the subroutine return address, where control is transferred on exit from this block.

Local Variables

This field contains all local variables for this block. It does not appear for blocks without local variables.

Static Link

The Static Link appears only in blocks which are lexically enclosed by other procedure or function blocks. The Static Link is used for references to intermediate level variables in the enclosing block(s). It points to the base of the Stack Frame of the latest invocation of the immediately enclosing procedure or function block, and it is the first link in the Static Link chain.

The Stack Pointer (SP) is also used for transient temporary storage, as in interrupts and Pascal library calls. Each For statement requires 3 words of temporary stack storage during its execution.

Embedded Assembly Code

PDP-11 assembly code can be embedded within an OMSI Pascal-1 program at any point where a comment might appear. Embedded assembly code takes the form of a special comment beginning with the embedded switch $C, as in the comment

```
{$C MOV %0,-(%6) }
```

The assembly code section extends to the closing comment brace (this closing brace cannot appear in an assembler comment). Any of the capabilities of the MACRO assembler may be used.

The OMSI Pascal-1 compiler scans the embedded assembly code and replaces tokens within the code which correspond to certain classes of Pascal identifiers. This provides simplified access to

Pascal data and control structures. However, the programmer is required to have some understanding of the internal structures. See the section on Runtime Memory Organization, and examine the code produced by the compiler.

Constant identifers appearing in assembly code are replaced by their defined values. Variable identifiers are replaced by the numeric offset from the appropriate base pointer. For global variables, the base pointer is Register 5 (R5); for local variables, the stack pointer (SP) is the base. For example, to swap the halves of a local integer variable I, the code would be

```
{$C SWAB I(SP) }
```

and to assign the constant Ten to the global variable Count one can write

```
{$C MOV #TEN,COUNT(R5) }
```

Any temporary stack usage is not recognized by the compiler, and must be included in indexed addressing of local variables.

Parameters of Pascal procedures and functions are treated as local variables, and are accessible in the same fashion. Internally, a Var parameter is the address of the actual parameter, so references to Var parameters must be indirect, as in

```
{$C MOV @VAR(SP),R0 }
```

Procedure and function identifiers are replaced by the internal label assigned by the compiler. To assign a value to a function, it is best to move the value to a local variable and then use a Pascal assignment statement to copy the value to the function.

The programmer is responsible for selecting the proper base register, as the compiler provides no error checking capability. Identifier substition is performed for all identifiers in these classes. This can cause problems if the programmer defines an identifier which corresponds to a MACRO operation, such as a constant named 'MOV'.

With one exception, the contents of registers R0-R4 may be changed within embedded code sections. Registers allocated for use in With statements must be preserved. With register allocation is in the order R3, R2, R1 and can be determined from the Pascal program. The contents of registers R5 and SP must always be preserved across the range of an embedded code section.

The default numeric radix of a Pascal-produced assembly code file is decimal, not the normal octal.

The System Error() Procedure

When a fatal runtime error occurs, the system procedure Error() is called with parameters describing the error and the system state. The Error() procedure is known by the global name ERROR, and 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 FCS 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 exiting from the Error() procedure normally results in program termination. The program global variables are available and may aid in diagnosing 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 on operating system facilites to 'chain' and restart the program or initiate another program.

Compiler Error Messages

',' used instead of ';'
8 or 9 in octal constant
Argument must be integer
Argument must be ordinal type
Argument must be real
ARRAY index out of range
ARRAY index type error
Bad ABS argument
Bad argument
Bad CASE label
Bad constant
Bad EXIT
Bad expression
Bad field list
Bad FILE name
Bad FOR statement
Bad FUNCTION name
Bad FUNCTION result type
Bad IN operands
Bad index type
Bad LABEL
Bad ORIGIN for variable
Bad parameter
Bad PROCEDURE name
Bad PROGRAM name
Bad READ statement
Bad RECORD
Bad scalar type
Bad SET element
Bad subrange
Bad TYPE
Bad TYPE specification
Bad variable list
Bad variant
Bad WITH statement
Bad WRITE statement
Boolean expression needed
Constant overflow
Don't repeat FORWARD parameter list
Duplicate CASE label
Duplicate field name
ELSE must be last in CASE
Expression too complex - out of registers
Expression too complex - out of registers (real)
Field list must be in parentheses
File variable missing
Format expression must be integer
FORTRAN must be VAR parameters
FUNCTION arg must be real or integer
FUNCTION argument missing

Illegal assignment
Illegal character
Illegal operator
Illegal type of operand
Improper symbol
Incompatible ARRAY type
Incompatible type
Invalid declaration, probably missing END
Invalid symbol
LABEL defined at wrong level
Label must be integer
LABEL not declared
LABEL redefinition
Local VAR definitions must precede PROCEDURE definitions
Missing ')'
Missing ')' at end of list
Missing '.' at end of program
Missing BEGIN
Missing END
Missing END in CASE
Missing field variable
Missing LABEL
Missing label definition
Missing operand
Missing operator
Missing semicolon
Missing UNTIL
Must be simple variable
NEW or DISPOSE arg must be pointer
Not implemented
ODD argument must be integer
Output file error
Source line too long
Strange '[' - bad SET or missing ARRAY definition
TEXT file expected
Too few arguments
Too many arguments
Too many errors in this line
Too many errors!
Too many levels
Too many symbols
Undefined FORWARD PROCEDURE or FUNCTION
Undefined operand
Undefined pointer base type
Undefined symbol
Unresolved forward type reference
WITH nested too deep

Runtime Error Messages

Can't open file
Compiler/Library mismatch -- Please recompile
Default file name syntax error
Default file switch error
Division by zero
Double deallocation of dynamic memory
Error reading file
Error writing file
EXP.overflow
File name syntax error
File not open
File switch error
Floating point format error
Floating point overflow
Illegal value for integer
Integer conversion error
Integer overflow
LOG of zero or a negative number
NEW of zero length
Not a random access file
Not enough memory
Reading past end of file
Reserved instruction trap
SEEK to record zero
Set element out of range
Square root of a negative number
Subscript out of bounds
Too many files open

OMSI Pascal-1 V1.2 Debugger (POD)

Contents

OMSI Pascal on-line debugging system documentation.

Introduction

The Pascal On-line Debugging system (POD) is a symbolic debugging tool that lets you interactively control the execution of your Pascal program. You can suspend execution at particular statements, execute one statement at a time, and examine and modify the values of particular variables. Since POD traps errors and identifies the last statement executed, you can easily pinpoint the source of run-time errors.

POD is really a series of Pascal procedures which are linked with a program. When you specify the debugging option (/D), the Pascal compiler includes a call to POD before each procedure and statement in your program. This lets POD control program execution. The compiler also produces a symbol table file containing the definitions and locations of all variables and procedures in your program. Using this, POD can find and modify variables and refer to procedures by name.

How to include POD in your program

To use POD, you must compile your program with the debugging switch, /D. You must also include a third output file in the compilation command -- this is the symbol table file for your program. For example:

```
>PAS TRIM,,TRIM=TRIM/D
```

The /D switch causes debugging instructions to be included in the compiled program. The third output file is a debugger file (TRIM.SYM) containing the symbol table information for the procedures and variables of TRIM.

POD supports an option called source debugging, selected using the /S compilation switch. This lets POD print the Pascal source lines associated with the compiled statements in your program. With the /S switch you can debug a program without having to print a listing of the program. The cost for using source debugging is an increase in the size of the program being debugged and a somewhat slower execution speed. All of the examples in this manual use the source debugging option.

If you wish to use the source debugging option, specify both the /S and /D switches in the compilation command, and include three output files (MACRO, listing, and symbol table):

```
>PAS TRIM,TRIM,TRIM=TRIM/S/D
```

POD reads the listing file file to display the source program for each Pascal statement. If the listing file is deleted, source debugging is automatically disabled, and POD will then identify statements only by procedure name and statement number.

POD itself is a large Pascal program (about 12K words) which resides in the same partition as the program being debugged. The procedures and functions of POD can be overlaid to reduce the memory requirements to about 4K words. Instructions for overlaying POD can be found in the Overlays section of the Programmer's Guide.

Running POD

When your program starts executing, POD will identify itself and ask you for the name of your program. It is assumed that the symbol file and listing file (if the S option is invoked) will share the program name. If either file cannot be found, POD will ask specifically for the necessary file name. If POD asks for a listing file and none exists, give a carriage return. This will cancel the source debugging option. POD will then ask for a symbol file name. Here is a typical POD opening dialogue:

```
>RUN TRIM
POD (Pascal On-line Debugger) - 24-Apr-79
POD - program name? TRIM
}
```

When POD is ready to accept commands, it will prompt you with a right brace (}). On some terminals this will print as a right square bracket (]). Commands to POD may be typed in either lower or upper case, and spaces in the commands are ignored. Several POD commands can be typed on the same line by separating the commands with semi-colons (;).

When you are finished with a debugging session, exit from POD by typing Ctrl/Z , or by typing two Ctrl/C's.

POD commands are presented alphabetically beginning on page 7.

Accessing Pascal statements

POD identifies Pascal statements by the name of the procedure containing the statement and the number of the statement in the procedure. The statement number can be found in the column labeled STMT in the listing file produced by the Pascal compiler. Statements in the main body of a Pascal program are considered to be in the procedure MAIN. All Pascal programs begin executing at MAIN,1. If the source debugging option is being used, POD will usually print the the source line along with the procedure name and statement number.

Pascal allows you to define procedures which define other local procedures. In this way it is possible to create a program containing several procedures all having the same name. It is strongly recommended that all of the procedures in your program have unique names in order to avoid confusion during debugging.

Accessing Pascal variables

POD lets you access the variables in your program in much the same way as you use variables in Pascal. Variables and procedure parameters are identified by name; such as MARGIN, LIMIT, or SHOESIZE. Records are specified using the standard dot notation such as: COORD.X, and RANGE.TOLERANCE.LOW. POD will generate an error message if too few (or too many) fields are specified for a record. Arrays of multiple dimensions are allowed, and POD will check the data type and limits of each index when accessing arrays. Pointers are specified in the usual way. The value of the pointer itself is interpreted as a decimal integer. A nil pointer has a value of zero, and POD will generate an error message if a reference through a nil pointer is attempted.

You can access very complex structures by combining several of the structures described above. In general, POD can access a variable in a structure in the same way as that variable is used in your program. Examples of legal variables are shown below:

```
FEET
A.B.C.D
CHIP^.TEMPLATE[3,1,-5].FLUX
PTR^.SON^.SON^.SON
```

Integers are treated as 16 bit signed numbers. Octal integers are specified by placing a "B" after the integer such as 377B. Boolean variables take values of either TRUE or FALSE. Character data, including character strings, are always enclosed within single quotes as with 'X' and 'THIS IS A TEST'. Spaces are not ignored within a character string. Real variables are used in the usual way. POD can also access scalar types defined by the user. For example, consider the program section below:

```
TYPE
    COLOR=(RED, WHITE, BLUE);
VAR
    X: COLOR;
```

When POD displays the value of X, it will correctly print the scalar type of X. This capability is provided only by POD -- Standard Pascal does not permit output of scalar types.

```
} X:=RED
} W(X)
RED
}
```

POD has another facility not available to the Pascal programmer: its ability to display the value of sets. The ".." notation for included set elements is available for both the input and output of set values.

```
TYPE
    COLOR=(RED, ORANGE, YELLOW, GREEN, BLUE);

VAR
    RB: SET OF COLOR;
    VALUES: SET OF INTEGER;
    Q: SET OF CHAR;
```

These variables may be accessed by POD as shown below:

```
} RB:=[RED..YELLOW,BLUE]
} W(RB)
[RED..YELLOW,BLUE]
} VALUES:=[1..20,50,40,30]; W(VALUES)
[1..20,30,40,50]
} Q:=['E','A','C','F','B','D']
} W(Q)
['A'..'F']
}
```

As demonstrated above, POD lets you assign values to variables in the same way as you assign values to variables in your program. The only restriction is that you cannot evaluate expressions such as C:=A+B, and you cannot call functions such as R:=SIN(3.1415).

POD enforces the Pascal scope rules. In general, this means that at any point in your program you can only access the variables that the program itself can access at that point. Global level variables, those defined at the start of the program, are always available. However, as different procedures are executed, the local variables and arguments of those procedures are temporarily available, while the local variables in procedures not being executed are never available. If you try to use a variable which is not available, POD will print a "symbol not found" error message. Remember, at any statement, you can only use the variables that are available to the program at that point.

POD lets you directly address memory locations as integers. For example, 1234B:=240B modifies location 1234 (octal) to contain 240 (octal). This feature is most commonly used when dealing with pointers. However, be careful, for you might accidently modify a location within your program and cause unpredictable results.

B(): Set/Clear Breakpoints

The "B" command sets a breakpoint at a particular statement within a program. Before POD executes each statement in your program it checks to see if a breakpoint has been set at that statement. If a breakpoint has been set, POD suspends the execution of the program and enters command mode. At this point you can examine and alter variables, check the history of the program's execution, or continue the execution of the program.

To set a breakpoint at a statement, type a "B" followed by the statement identifier (procedure and statement number) contained within parentheses. POD will interrupt the execution of your program just before the statement at which a breakpoint is set. Up to eight breakpoints may be in effect at any one time. Examples:

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B(INIT,5); C
Breakpoint at INIT,5  PARAM1:=0; PARAM2:=0;
}
```

(The examples above show how the source debugging option works. When POD stops at breakpoint, it prints the Pascal source line for that statement.)

The "G" command in the example starts program execution. The "C" command continues from the breakpoint.

If your program "runs away" or loops unexpectedly, you may regain control at any time by typing Ctrl/C -- this causes an immediate breakpoint interrupt and returns control to the Debugger.

POD has the capability to execute a series of POD commands when a breakpoint is encountered. This facility, called stored commands, is specified by placing the command within angle brackets (< >) after the break command as shown here:

```
} B(MAIN,6) < W(DEPTH); DEPTH:=5 >
} B(POSITION,32)<W(X,Y);C>
```

The first example displays the value of the variable DEPTH then assigns the value of 5 to DEPTH each time the program comes to the statement at MAIN,6. The second example displays the values of the variables X and Y and then continues the execution of the program. In this case POD will not stop and enter command mode.

Instead, each time the program comes to the statement at POSITION,32, the variables X and Y will be displayed and the program will continue.

Any POD command may appear in a stored command, but stored commands may not be nested, ie. a stored command may not define other stored commands. As many POD commands as will fit on a single line may be specified in a stored command.

There are two ways to cancel a breakpoint. The "K" command described below can be used to kill all breakpoints or just a single breakpoint. However, if the program has just been interrupted because a breakpoint was reached, that breakpoint can be cancelled by using the "B" command with no arguments.

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B
} C
```

The "D" command may be used to display the currently active breakpoints and their associated stored commands.

C, C(): Continue execution

If the execution of your program has been suspended by POD, you may use the "C" command to resume execution of the program. If your program has not started executing, either the "C" or the "G" command may be used to start the program. The section above describing breakpoints has several examples which use the "C" command. Once your program has terminated and POD has re-entered command mode, any attempts to continue the program with the "C" command will be ignored. (There is nowhere to go!) The program may, however, be restarted with the "G" command described below.

If you set a breakpoint inside a loop, it is sometimes desirable to let the statement at the breakpoint execute several times before stopping. One way to do this is to use the "C" command several times to continue from the breakpoint until the desired iteration in the loop is reached. Another solution is to use a repeat count contained inside parentheses after the "C". The repeat count tells how many times the statement at which the breakpoint has been set should be executed before the breakpoint takes effect. For example, you can set a breakpoint at COUNT,10 which is inside a loop structure. When the loop is first entered, POD will stop the program at COUNT,10 with a breakpoint. The command C(6) will let the loop iterate 6 times before the program stops again at COUNT,10 with a breakpoint. Each of the eight breakpoints has its own repeat count.

D: Display POD Parameters

The "D" command displays the watched variables, labels, and breakpoints which are currently active. Watched variables are described below in the section about the "V" command. Labels are discussed below in the sections about the "G" and "L" commands. The stored commands associated with breakpoints and the watched variable are also displayed.

```
} D

Watching: B[5] <W(B[6],B[7],B[8])>

Breakpoints:
MAIN,13 <W(FOO);C>
MAIN,20
ERR,5   <W(ERRORCODE);H>

User defined labels:
1:  MAIN,1  BEGIN I:=0;
5:  RETRY,3  RESET(F,NAME,'DAT',STATUS);
}
```

G, G(): Go or Go to a Label

The "G" command without arguments starts or restarts your program at MAIN,1. If the "G" command is followed by a label number in parentheses, the program will be continued at that user defined label. Do not confuse user defined labels with Pascal statement labels. User defined labels are created with the "L" command dynamically as POD controls your program. Pascal statement labels are defined in your source code and are used by the PASCAL compiler to generate targets for the Pascal GOTO command. POD does not use Pascal statement labels.

The "L" command labels the program statement about to be executed. The most common way to define a label at a particular statement is to set a breakpoint at that statement, execute the program until that statement is reached, and then use the "L" command to define the label.

The "G" command should be used with care. It is not always possible to branch from any Pascal statement to any other Pascal statement. Labels follow the same scope rules as variables, so depending on which procedures are being executed, some labels may not be available. If you try to go to a label which is not available, POD will respond with the error message "You can't get there from here". One reason that POD cannot go to a particular label is that if the label is in a procedure which is not being executed, POD is not able to invent the values of the local variables associated with that procedure.

```
} B(MAIN,5); C
Breakpoint at MAIN,5  J:=SIN(Q);
} L(3); B(MAIN,27); C
Breakpoint at MAIN,27  WRITELN('X>Y');
} G(3)
Breakpoint at MAIN,27  WRITELN('X>Y');
} G
Breakpoint at MAIN,5  J:=SIN(Q);
}
```

H: Print Program Execution History

POD maintains a list of the last 10 statements executed by a program. This history is useful in determining how the program got to a breakpoint or how it got to a statement which caused an error. The "H" command prints the history and also the procedure execution stack. The stack shows the procedure and function nesting all the way back to the main body of the program.

```
} B(EVALUATEBOARD,1);C
Breakpoint at EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
} H
Program execution history

GENMOVE,3  BEGIN
GENMOVE,4  FATHER:=F;
GENMOVE,5  MOVE:=I*256+J;
GENMOVE,6  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,7  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,8  IF TURN=BLACK THEN
GENMOVE,9  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,11  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;

Procedure execution stack

EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
MOVEPIECE,11  IF MOVESALLOWED THEN GENMOVE(I,J);
EXPAND,15  IF COLOR[WHO]=TURN THEN MOVEPIECE(I,I,0,0);
MAIN,7  EXPAND(ROOT,TRUE);
}
```

K, K(): Kill Breakpoints and Labels

When the "K" command is given without arguments, all label definitions and breakpoints are deleted. When the "K" command is followed by a statement identifier, the breakpoint at that statement is removed.

```
} B(MAIN,5)
} K(MAIN,5)
} B(MAIN,17)
} K
```

Individual breakpoints can also be removed with the "B" command.

L(): Label a Statement

You may label up to eight statements with the "L" command. Labels are used as targets of the "G" command. The label number (1 through 8) is placed in parentheses after the "L". The "L" command always defines the label at the current location within the program being executed. Check the description of the "G" command above for a warning about branching within a Pascal program. The "D" command may be used to list the currently active labels.

```
} B(MAIN,13); G
Breakpoint at MAIN,13  A:=1;
} L(1)
} B(MAIN,15); C
Breakpoint at MAIN,15  B:=37;
} L(5)
} D

Breakpoints:
MAIN,13
MAIN,15

User defined labels:
1:  MAIN,13  A:=1;
5:  MAIN,15  B:=37;
}
```

P, P(): Execute one Statement in Current Procedure

The "P" command executes a single statement in the current procedure. "P" will not single step through functions and procedures nested in the current procedure, but instead will treat their calls as single statements. If the current procedure ends, "P" will begin single stepping the procedure that called the current procedure. (Compare "P" to the similar "S" command described below.)

If a repeat count is given in parentheses after the "P", the specified number of statements will be executed before stopping. As with the "C" command, you may not proceed past the end of the program once the program has terminated. Use the "G" command to restart the program.

```
} P
Breakpoint at MAIN,1  BEGIN I:=0;
} P
Breakpoint at MAIN,2  J:=RANDOMINTEGER(3);
} P
Breakpoint at MAIN,3  K:=J*J-I;
} P(5)
Breakpoint at MAIN,8  IF K<J THEN BEGIN
}
```

R: Register Dump

The "R" command prints the values of the processor registers R0-PC in both octal and decimal. This command is normally useful only to those programmers who include in-line assembly language code in their Pascal programs.

S, S(): Single Step

The "S" command is identical to the "P" command above, except that if a statement being stepped through contains a procedure or function call then the new procedure or function will be executed one step at a time. As with "P", a repeat count may be specified.

```
} S
Breakpoint at MAIN,1  BEGIN I:=0;
} S
Breakpoint at MAIN,2  RANDOMINTEGER(3);
} S(1)
Breakpoint at RANDOMINTEGER,1  BEGIN RANDOM:=X;
}
```

T(): Trace Mode

"T(TRUE)" turns on statement trace mode, while "T(FALSE)" turns it off. When trace mode is on, POD will print the location of each statement before it is executed. If several PASCAL statements appear on the same line in the source file, and if those statements are each executed in sequence, then the line containing those statements will be printed only once.

```
} B(MAIN,6)
} T(TRUE)
} G
MAIN,1  BEGIN I:=0;
MAIN,2  J:=0; K:=0; L:=3.14159;
MAIN,5  WRITELN('HI THERE');
HI THERE
Breakpoint at MAIN,6  WRITELN;
}
```

V(): Variable Watch

The "V" command makes POD watch the value of a variable. Before each statement in your program is executed, POD compares the current value of the variable with the value it had when the "V" command was given. If the value has changed, POD stops your program and tells you so. If you continue your program, POD will continue watching for a change in the variable.

The "V" command is useful if your program is malfunctioning because the value of some critical variable is being destroyed somewhere. The "V" command can also be used to watch locations in low memory to detect the incorrect use of a nil pointer.

```
} V(DEPTH)
} C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 0
New value: 1
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
} C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 1
New value: 2
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
} C
Value of "DEPTH" changed at statement:
DESCEND,38  DEPTH:=DEPTH-1;
Old value: 2
New value: 1
Breakpoint at DESCEND,39  END;
}
```

Stored commands may be specified with the "V" command in the same way as with the "B" command. The "D" command will list the name of the variable being watched and the stored commands if any were given. A variable watch is terminated by using the "V" command with no arguments. POD will automatically terminate a watch on a variable when that variable is no longer available. When POD does this, it prints the message "Watch terminated -- value didn't change".

```
        } B(EVALUATEBOARD,35); C
        Breakpoint at EVALUATEBOARD,35  FOR I:=5 TO 39 DO
        } V(BLACKSCORE)<W(WHITESCORE)>
        } C
        Value of "BLACKSCORE" changed at statement:
        EVALUATEBOARD,224  ELSE BLACKSCORE:=BLACKSCORE+MOC4;
        Old value: 0
        New value: 400
        Breakpoint at EVALUATEBOARD,225  IF BLACKDENY<WHITEDENY
THEN
        0
        } C
        Watch terminated -- value didn't change
        Breakpoint at MAIN,28  MAXLEVEL:=0;
        }
```

W(): Write Variable Value

The "W" command is used to write the value of a variable, pointer, constant, or memory location. The format of the output is determined by the type of the variable being written. For example, integer variables are written as 16 bit signed decimal integers, while set variables are written using set notation. The names of the variable to be displayed are placed inside parentheses following the "W". If more than one variable is to be written then the names are separated by commas. Physical memory locations are addressed as integers (either octal or decimal). As in Pascal, integer and real values may use format control with the colon (:) notation. This is also how one examines memory locations in octal.

```
} W(TURN)
BLACK
} W(COLOR[BLACKKING],COLOR[WHITEKING])
BLACK
WHITE
} W(USERMOVES[5])
'B1'
} W(ROOT^.SON^.VALUE)
402
} W(54B)
-10154
} W(54B:-1)
154126B
} W(S)
['A','M','Z']
} W(R)
 3.141593E+00
} W(CH)
'A'
} W(I)
123
}
```

Advanced Debugging Techniques

If you write large Pascal programs, you might find that you are not able to use POD (even overlaid) to help debug your program because of memory size restrictions. However, there are several things you can do to further reduce the amount of memory required by POD. The easiest thing to disable source debugging. The use of the source debugging option (/S) expands your program by one word for every Pascal statement in your program. For large programs you may save more than 1K words by not using source debugging.

Another technique you can use is selective debugging. You can edit your program to turn off the generation of POD debugging information around procedures which have already been tested and debugged. To turn off debugging, place the line {$D-} before the procedure definition and {$D+} after the procedure. You will not be able to set breakpoints or examine variables in such procedures, but you will save two or three words for every statement not debugged. Be sure debugging is enabled around all variables you may wish to examine and around the main procedure.

If you program uses overlays, you can still debug your program using POD. When you compile the main body of the program, which resides in the root segment, use the debugging switch (/D) and produce a symbol table file. Compile each of the external modules in the normal way without the debugging switch. You cannot enable debugging in external procedures because you would have to produce a symbol table file for POD. The main body of your program must also have a symbol table, and there is no way to combine the two into a single usable file. The only way to debug an external procedure is to include its definition in the main program. In other words, you must make the procedure not be external.

When you task build your overlaid program you will have to use two overlay regions to contain the modules of POD. These two overlay regions may, in most cases, also contain your own external procedures. There should be no conflicts because POD only lets you debug in the root segment, and as long as the two POD modules RTDBG and DBG are placed in the root, there should be no problems with the overlays.

You cannot set breakpoints within external procedures, but you can cause a break when the external procedure is called from the main program. This is done by setting a breakpoint and giving only the name of the procedure at which to break as with: B(OVER1). This type of breakpoint will stop the program before the external procedure OVER1 is executed. The only variables you will be able to examine and modify in OVER1 are those variables in the parameter list for OVER1. Note that the names of the parameters are defined by the external procedure definition of OVER1 in the main program, not by the definitions in OVER1 itself.

OMSI Pascal-1 V1.2/RSX Installation Procedures

How to Install OMSI PASCAL-1 on RSX-11M

This release kit contains all the files necessary to build an OMSI Pascal-1 compiler and runtime library on any PDP-11 computer running RSX-11M.

First, login to any privileged UIC, and assign LB: as SY: so that the Pascal compiler and runtime library will be available to all users from LB:. Then, copy the command file called PASBLD.CMD from the distribution medium to your disk using one of the following commands:

```
>FLX SY:/RS=MT:[1,1]PASBLD.CMD/DO     (for magtape)

>FLX SY:/RS=DK:PASBLD.CMD/RT          (for RK05 disk)

>FLX SY:/RS=DX:PASBLD.CMD/RT          (for floppy diskettes)
```

Next, initiate the build process by executing the command file PASBLD.CMD. To do this, type:

```
>@PASBLD
```

You will be asked several questions to determine the hardware configuration of your system. Then, two object module libraries (a compiler library and a runtime library), a Task Builder command file, and the compiler ODL file will be selected from the distribution medium and copied onto the system disk. The Pascal compiler will be built and installed as ...PAS, and the runtime library will be copied to LB:[1,1]PASLIB.OLB. Finally, the build files which are no longer needed will be deleted.

After completing the automatic installation procedure, you should copy the utility programs, documentation files, and demonstration programs to your system. A description of the files on this release kit appears below.

LISTME.DOC This document.

PASBLD.CMD The MCR command file to build the Pascal compiler on your system and select one of the runtime libraries below.

PASCAL.CMD The Task Builder command file to build the Pascal compiler. Some compiler characteristics (such as symbol table size) can be changed by editing this file before installation.

PASCAL.ODL The overlay description of the Pascal compiler.

CMPFPP.OLB
CMPFIS.OLB The object module libraries for the Pascal compiler(s). CMPFPP is the compiler for processors with the FPP instruction set; CMPFIS is for all other processors.

FPPLIB.OLB
FISLIB.OLB
EISLIB.OLB
SIMLIB.OLB The Pascal runtime libraries. FPPLIB uses the FPP and EIS instruction sets; FISLIB uses the FIS and EIS instruction sets; EISLIB uses only the EIS instruction set, and SIMLIB operates on any PDP-11 processor.

PAS.ODL An overlay description to assist in overlaying the Pascal runtime library and Debugger to conserve memory. This file should be installed in LB:[1,1].

PROFIL.PAS The Pascal Profiler -- produces an annotated listing of a Pascal program showing how often each statement was executed. This file should be compiled, assembled, and made available to all users as LB:[1,1]PROFIL.OBJ.

FORMAT.PAS A Pascal program reformatter and cross reference generator. FORMAT should be compiled and available to all users.

IMP.PAS A post-compilation optimizer which performs branch optimization of the output of the OMSI Pascal compiler. This program should be compiled and made available to all users.

STRING.PAS A collection of Pascal procedures and functions which implement dynamic character string operations.

ERROR.PAS Source code for the standard error routine in the Pascal support library. ERROR is called whenever a runtime error is detected in a Pascal program.

FDB.PAS A modified version of the standard error procedure which prints a detailed map of the FDB associated with the file which caused the I/O error.

RANDOM.PAS A random number generator.

PL0.PAS Simple compiler example.

MAZE.PAS Demonstration maze generator and solver.

CHECKR.PAS Plays the game of checkers.

Programming Changes in OMSI Pascal-1 V1.2

There are four specific language features that have been changed from V1.1 to V1.2, all of which are related to I/O characteristics. If a program which was written using V1.1 fails to operate properly with V1.2, check the following trouble shooting points.

(1) In V1.1, Eoln() on interactive terminal files had the initial value True. This has changed to False in V1.2.

Symptom: program "hangs", or ignores the first input line.

Cure: remove the initial Readln(), or replace it by the statement "if Eoln() then Readln()", which runs correctly with either version.

(2) The V1.1 Read() procedure, when reading a (packed) array of Char, ignored leading blanks and terminated on a blank or a comma. The V1.2 Read() procedure will read characters without skipping blanks, and terminates at Eoln() or upon filling the array.

Symptom: program does not interpret commands properly, or loops.

Cure: reprogram sections which use the V1.1 Read(). Programs which are heavily dependent on the V1.1 style Read() may be more easily recoded using the V1.2 string package.

(3) The declaration "file of Char" is no longer equivalent to the declaration "Text". This change corresponds to the more strict type checking of the draft ISO Pascal Standard.

Symptom: compiler error message "TEXT file expected".

Cure: substitute the type Text and recompile.

(4) The Seek(), Deposit(), and CloseRandomFile() procedures which are supplied with V1.1 have been superseded by the built-in procedure Seek().

Symptom: unpredictable I/O failures. The V1.1 procedures will compile under V1.2, but will not operate correctly.

Cure: reprogram affected sections using the built-in Seek(). Note that the V1.2 Seek() numbers records beginning at 1.