OS-9 Pascal V2.0 User's Manual

OS-9 Pascal V2.0 User's Manual

Revision F

Publication date July, 1984

Chapter 1. An Overview of the Pascal System

About OS-9 Pascal

In the last few years, the Pascal language has become one of the most popular computer programming languages. Although it was originally developed as an aid in teaching computer science, it has found wide use in almost every imaginable computer application for good reason. The Pascal language gives programmers an almost perfect medium for concise expression of solutions to complex problems.

The internal operation of a language as powerful as Pascal must be relatively complex, and running Pascal programs can be quite demanding of the computer. Therefore, microcomputer versions of Pascal traditionally have been quite limited and much slower than their big computer cousins. The gap has been narrowed considerably in OS-9 Pascal because of two factors. The first factor is the 6809 microprocessor, which was specifically designed to efficiently execute high-level languages such as Pascal. The second factor is the part of OS-9 Pascal called “PascalS”, which allows the Pascal system to utilize disk space as “virtual memory”. Being able to utilize disk space as “virtual memory” means that you can run Pascal programs that are much larger than the actual memory size. Indeed, a Pascal compiler as complete as the OS-9 Pascal compiler would otherwise be too big to fit in your computer's memory.

One other unusual characteristic of OS-9 Pascal is its ability to compile and run programs in either “P-code” or “native code” forms. “P-codes” are instructions particularly created for an ideal, imaginary “Pascal Computer”. The 6809 can't directly execute P-code instructions, so a program called a “P-code interpreter” is used to simulate the ideal “Pascal Computer”. Most microcomputer versions of Pascal use the P-code concept because it simplifies the design of the compiler and makes most efficient use of a limited amount of memory. Another plus for P-code is that while programs are running, the P-code interpreter can perform thorough error checks and can give excellent diagnostic messages.

Using P-code, the execution speed of programs is relatively slow compared to true machine language. Each P-code instruction causes actual machine language instructions to be run in the interpreter program (which are “overhead” and not actually needed to carry out the original Pascal program). OS-9 Pascal provides a unique solution to this problem by means of a program called a “native code translator”. The native code translator takes a P-code program and translates it to 6809 assembly language (machine language) source code. Both the P-code and the native code forms of the program work exactly the same way - except that the native code version will run from four to ten times faster! And because the output of the translator is a text file that is processed by the standard OS-9 assembler, you can examine or manually edit it if you wish.

Why even bother with P-code? One reason is that very big programs will only fit in your computer in P-code form. While P-code is not as fast as native code, in many cases speed is not an important enough factor to bother with the optional translation step. Perhaps the main value of P-code is its use in program debugging because the P-code interpreter has more comprehensive error checking and diagnostics. Typically, OS-9 Pascal users debug programs in P-code form and translate to native code as a final step.

The steps in creating and testing a program in OS-9 Pascal are listed below:

1. Create the Pascal source program using a text editor.
2. Compile the program to P-code using the compiler.
3. If there are compilation errors, edit the Pascal source file and go back to step 2.
4. Run the program using the P-code interpreter.
5. If there are run-time errors or program errors, edit the Pascal source file and go back to step 2.

6. (Optional) Translate the P-code program to native code, then run the assembler with the native code source file.

Do You Know Pascal?

Either you already know Pascal, or you don't.

If you already know Pascal, you will be pleased to discover that OS-9 Pascal is a very thorough implementation of the language according to ISO Standard 7185.1 Level 0 with exceptions and extensions documented in Chapter 14 of this manual. The compiler behaves as the Wirth and Jensen “bible” says it should. You will discover that a number of very useful additional library functions that are not included in the ISO specification have been added to OS-9 Pascal. Also, some unnecessarily restrictive Pascal syntax requirements have been relaxed.

If you don't already know Pascal, you have some studying to do. Fortunately, Pascal was originally designed for teaching programming, so it is easy to learn in stages. Unfortunately, a course in Pascal programming is beyond the scope of this manual. The books listed on the next page are recommended as reference and self-study source books. They are generally available at, or through, many larger bookstores, or they can be ordered directly from the publishers.

Suggested ISO Pascal Source Books

Programming in Pascal, Revised Edition by Peter Grogono, Addison-Wesley Publishing Co., Reading, Mass., 1980

This book presents a good self-study course on Pascal for beginners. It is based on the ISO Pascal Standard which is important for compatibility with OS-9 Pascal. There are a large number of similar volumes or bookstore shelves, including many good self-study courses. Make sure any such book you may select is based on “Wirth/Jensen” or “ISO Standard” Pascal - but NOT the “UCSD Pascal” dialect, or you may have trouble running example programs exactly as given.

Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth, Springer-Verlag, New York, 1974

This book is the “bible”, written by the creators of the language itself. The first part of the book is a “user manual” that shows how Pascal programs are constructed. The second part is a “report” giving a concise description of Pascal's syntax. An invaluable reference work. When expert Pascal programmers argue over trivial points, this book is consulted to settle matters.

Standard Pascal User Reference Manual by Doug Cooper, W.W. Norton & Co., New York, 1983

The ISO Pascal Standard document was written by and for computer scientists; therefore, it is almost impossible to read or understand. This book does an admirable job of translating the ISO specification for those “with only human powers of understanding”, as Mr. Cooper puts it. Fortunately, the result is still technically accurate. You may find this to be a better reference than the Wirth/Jensen classic.

Requirements and Information

Requirements:	64K Color Computer
	2 disk drives
	2 OS-9 Pascal distribution disk

Before using your Pascal, it is important to create a BACKUP of your two OS-9 distribution disks. The original disks should then be stored in a safe place. To protect the information that you store on diskette, make backups of your working disks frequently.

The steps two take to create backup disks are as follows:

1. FORMAT a new disk.

Place a disk in drive 1 and type FORMAT /D1. The format utility will print a ready prompt. Type 'y' and the disk will be formatted.

2. BACKUP the disk.

   For a two drive backup, place the source disk in drive 0, a formatted disk in drive 1, and then type BACKUP. The backup utility will prompt for any input.

Each Pascal disk is a Minimum System Configured disk. Only necessary OS-9 commands have been placed on the disks. When creating backup Pascal disks you may add or delete commands if space allows.

The Parts of the Pascal System and Installation

Your copy of the OS-9 Pascal system will consist of two diskettes. Before you use Pascal, you must copy these files onto your system disk and a working disk (or directory).

You should create a special system disk for working with Pascal that has only the minimum set of commonly used OS-9 commands (such as COPY, DEL, DIR, etc.) and omits those commands not frequently used when working with Pascal (such as FORMAT, DCHECK, OS9GEN, etc.). The two diskettes containing OS-9 Pascal are already setup as minimal system disks and may used as examples.

To use your special system disks, simply boot you system with your normal system disk then insert you special system disk in drive 0 and type CHX /D0/CMDS which will now use the CMDS directory on the special system disk as your new execution directory. You should then use the CHD command to change your current data directory to the directory in which you wish to create source files. The OS-9 Pascal disk number 1 contains all programs required to compile and execute pascal programs using one of the interpreters.

The table below lists the files included in the OS-9 Pascal System. The type code indicates the type of data stored on the file: M=6809 machine language, P=Pcode, and T=text. Each file must be copied from the distribution disk to either your system's execution ("CMDS") directory or to a directory or disk reserved for Pascal use, thus the last column shows where the file is to be copied. Directory names with asterisks indicate that the use of the file is optional if disk space is limited.

File Name	Function	Type	Directory
Pascal	Pascal compiler program	M	CMDS
Pascal_Compiler	Pascal compiler pcode file	P	CMDS
PascalErrs	Text file of error messages	T	CMDS
PascalN	Normal pcode interpreter	M	CMDS
PascalS	Swapping pcode interpreter	M	CMDS
PascalT.PRUN	Native code translator pcode file	P	CMDS
PascalT.MODL	Native code external routines used by PascalT.PRUN	M	CMDS
PascalE	External routine mapping program.	M	CMDS
PascalDefs	assembly language equates for native code programs.	T	PASCAL
Support	Full support package module.	M	CMDS
Support1	Support package module without SIN, COS, LN, EXP, ATAN, and SQRT	M	CMDS*
Support2	Same as Support1 except without real numbers, AFRAC,	M	CMDS*

File Name	Function	Type	Directory
	AINT, FILESIZE, CNVTREAL, SEEKEOF, real READ/WRITE		
DumpReal	Sample source program	T	PASCAL

Pascal. "PASCAL" is the native code portion of the compiler which prepares for execution using "Pascal_Compiler", which is the main body of the compiler in P-code form. Chapter 2 of this manual describes how to run the "Pascal" program.

PascalErrs. PascalErrs is a text file containing error message strings. The compiler, P-code interpreters, and Support use the text file "PascalErrs" to generate full English error messages- See page 3-3 for more information.

PascalN and PascalS. PascalN and PascalS are the P-code interpreters. "PascalN" is the "normal" (and faster) interpreter used to run compiled P-code programs. "PascalS" is a swapping interpreter that uses a temporary disk file to simulate program and data memory so very large programs can be run. Because of its size, the native code translator is run by PascalS. PascalS can also be used to run user programs. See Chapter 3 for details.

PascalT.PRUN and PascalT.MODL. "PascalT.PRUN" is the native code translator program which is run using the PascalS swapping P-code interpreter. PascalT.PRUN is a large P-code program that uses some machine language procedures to improve its translation speed which it automatically loads from the file PascalT.MODL. See Chapter 4 for details.

PascalE. "PascalE" is an linkage editor program used to link into a single program separately compiled procedures having EXTERNAL declarations. See Chapter 5 for details.

PascalDefs. PascalDefs is a text file containing assembly language source code definitions required when assembling programs produced by the native code translator.

Support, Support1, and Support2. "Support" (10k) is a machine language module containing library routines commonly used by all components of the OS-9 Pascal system. It includes subroutines for floating point arithmetic functions, input/output operations, the operating system interface, etc. It is used by PascalS, PascalN, and native code programs generated by the translator. "Support1" (8k) and "Support2" (6k) are stripped down versions of "Support" that can be used to save memory if certain library functions (such as transcendental functions) are not required by specific applications.

Chapter 2. “PASCAL” - The Pascal Compiler

SYNOPSIS: The function of the compiler is to convert a Pascal “source” program file to a P-code file that may be executed using one of the interpreters. Running the compiler is always the first step in preparing a Pascal program regardless of whether you intend to produce a P-code program or a native code program.

The Compiler Command Line

The command line consists of the command “Pascal”, The source file is read from the standard input path, so the OS-9 Shell redirection operator “<” is used to redirect input of the compiler to the file desired. A typical command used to call the compiler looks like this:

```
PASCAL  <myprogram  #28k
```

The PASCAL command compiles a program on a file called “myprogram”. The OS-9 Shell memory size option is used to give the compiler 28K bytes of working memory. The compiler will display a program listing on the terminal display (the standard output path) and write the corresponding P-code program on a file called “PCODEF”.

After the compilation has finished, the normal P-code interpreter called “PascalN” can be used to run the program on the PCODEF file. For example:

```
PASCALN PCODEF
```

Compiling and running Pascal programs is often as simple as this. On the other hand, the Pascal compiler is a very sophisticated program that has many user-controllable features that you may want to use as a matter of preference, or for special reasons related to the program you are developing. This chapter is devoted to a detailed description of the compiler's many optional features. For more information on PascalN, See Chapter 3.

The Pascal Program Source File

The source file containing the program to be compiled is prepared using the system text editor on your minimum configuration OS-9 disk, or the sample program furnished with the OS-9 Pascal System can be used to familiarize yourself with how the compiler works. The source file generally resides in the “PASCAL” working directory or disk.

In preparing your source program, keep the following rules in mind:

- Only the first 110 characters on a line will be recognized.
- String constants (sequences of 2 or more characters between single quote marks) can have up to 100 characters.
- The program itself can contain up to 254 procedures including procedures declared as EXTERNAL.

A Detailed Description of Compiler Command Lines

The compiler is called using a command line of the form:

```
PASCAL <sourcefile >listfile >>statfile  #memsize : parameters
```

Note that PASCAL uses OS-9's standard I/O paths for normal input and output operations which may be redirected to or from any file or device.

The source file is read from the standard input path ("<sourcefile"). Normally, this is a disk file.

The compiler listing is written to the standard output path (">listfile"). If the listing output is not redirected, the listing will be displayed on the user's terminal. At the end of the listing, the compiler will display a procedure table. Also, if routine errors are detected during compilation, the source line causing the error and the appropriate error message(s) will be shown on this path. You can, however, via either a compiler option or a parameter (see below), inhibit the listing of the source program while it is being compiled.

The standard error/status path (">>/statfile") is used for compilation statistics and severe error messages. If no redirection of this path is specified, messages will be displayed on the user's terminal. Severe error messages report conditions which prevent the compiler from operating, such as insufficient memory or OS-9 file errors. “Normal” error messages are displayed on the listing path described above. Compilation statistics show figures for various types of memory usage which can be used to improve compiler throughput for the particular program being compiled.

The “#memsize” specification determines how much memory the compiler is given by OS-9. The larger this number is, the faster the compiler will run. Therefore, you should use the largest amount possible on your system. The amount of memory is best determined by experimentation. Most OS-9 systems allow the amount to be at least 60 to 80 pages (15K to 20K). If you receive an “ERROR 207” message from OS-9 after you attempt to run Pascal, it means that the memory size you specified was too large.

Here are some sample command lines:

```
pascal <fibonacci >/p >>/p #18K
pascal <invert >/p : d50 t
```

Command Line Parameters

A list of one or more parameters for the compiler can optionally be included in the command line. If parameters are used, they must be preceded by the “:” character. The parameters allow global control over program listing functions and over certain code generation options. They also allow redirection of the P-code file produced by the compiler and/or of the compiler work, which can be useful in a multi-user environment.

Spaces or commas can be used to separate parameters, but the total number of characters in the parameter list must be less than or equal to 80, including all space and comma characters. Upper and lower case letters are considered to be equivalent. The options allowed, their form, and their effect are as follows:

O=<*pathname*>	“`O`” will change the name of the PCODEF file produced by the compiler to “pathname”.
T=<*pathname*>	“`T`” will change the name of the PROCTAB file produced by the compiler to “pathname”.
T	“`T`”, used by itself, will inhibit the printing of symbol table dumps. “`T`” is a global inhibit and overrides any symbol table dump options provided within the source program {see the section on compiler options below).
L	“`L`” inhibits the source program listing. “`L`” is a global inhibit and overrides any listing options provided within the source program (see the section on compiler options below).
F	“`F`” will inhibit the use of a form feed character to advance to a new page. Instead, carriage return/line feed combinations will be used to space to a new page. The compiler always begins printing at the current position of the listing device and always issues a new page sequence at the end of every compilation.

D<*number*>	"D" followed by a number greater than 11 will set the page length to that number. Nine lines of source text less than this number will be printed on each page. Three of the nine lines are occupied by the heading for each page, and at the end of each page, six blank lines are printed . If not overridden, the default page length is 66. If the number given is less than 12, it will be ignored, and 12 will be assumed. If the number given is greater than 160, it will be ignored, and 160 will be assumed.
D	"D" by itself inhibits the generation of run time range checks for array indexing, memory references via pointers, and assignments to Boolean, set, or subrange type variables. "D" is a global inhibit and overrides any range check options provided within the source program (see the section on compiler options below and the Debug section of Chapter 9).
W<*number*>	"W" followed by a number greater than 16 will set the maximum line width to that number. Note that "W" affects the listing of source text lines — the three lines of heading for each page have their own minimum length which may be longer than this value. Source text lines which would require a longer line to be totally listed are simply truncated for listing purposes. Each line of source text is preceded by a 16 character header (see the section below on reading the compilation listing), so that a line width of 17, the minimum allowed, would result in only the first character of every source line being listed. If not overridden, the default line width is 79. If the number given is less than 17, the number will be ignored, and 17 will be assumed. If the number given is greater than 160, the number will be ignored, and 160 will be assumed.
N	"N" will inhibit the inclusion of source line numbers in the P-code file. "N" is a global inhibit and overrides any line number options provided within the source program (see the section on compiler options below). Inclusion of line numbers will cause run-time error reporting to report source line numbers as an aid in program debugging. Line numbers require three bytes of P-code for nearly every statement in the source program, and program execution is slowed slightly. Therefore, this option should be given when compiling "final" versions of programs when no further debugging is anticipated.

Compile-Time Options

Compile-time options may be included inside Pascal source programs. They are used to enable or disable various compile time modes or to override certain memory allocation default values. These options use a form similar to comment lines:

`(*$ optionlist *)` or `{$ optionlist }`

When either of these forms is encountered by the compiler, the text inside the comment is processed as a list of one or more compile-time options. If the compiler fails to find proper syntax, then the whole string is treated simply as another normal comment, and no message is given. More than one option can be specified within a single comment as long as each has the proper syntax and are separated by a comma and optional spaces. For example, if three good option specifications were found within the comment and then some sequence of characters was found which wasn't recognized as a valid option specification, the three preceding specifications are properly kept and the remainder of the comment is ignored. Upper and lower case letters are considered equivalent. The options, their form, and their function are as follows:

Here are some examples:

```
(*$l-          Turn off listing *)
(*$l+,T+       Enable listing and symbol table dump*)
{$l10000, e1000 Increase size of local and extended
               stacks}
```

```
{$S20480        Request 20K for swap buffers for
                PascalS}
```

`l+` *Enables program listing.

`l-` Disables program listing.

`l` Followed by a number overrides the compiled program's local stack size calculation. See Chapter 10 for more information.

`e` Followed by a number overrides the compiled program's extended stack size calculation. See Chapter 10 for further more information.

`h` Followed by a number overrides the compiled program's minimum heap size calculation. The compiler always calculates the minimum heap size to be zero bytes unless overridden. See Chanter 10 for more information.

`s` Followed by a number overrides the compiled program's default swap size calculation. The compiler always calculates the minimum swap size to be 2K bytes unless overridden. This value, as with the other size values just described, only affects memory allocation for the running of the compiled version of this program - it does not in any way affect the process of compiling the source program. Also, if any memory size option is given more than once, only the last specification is kept. See Chapter 10 for more information.

`d+` *Enables generation of run time range checks for array indexing, memory references via pointers, and assignments to Boolean, set, or subrange type variables. Case statements are always checked. The “d” option is also referred to as the “Debug Option”. See Chapter 9 for more information on the Debug Option.

`d-` Disables generation of run time range checks described above.

`n+` *Enables inclusion of source line numbers in the P-code file.

`n-` Disables inclusion of source line numbers in the P-code file.

`t+` Enables the generation of symbol table dumps.

`t-` *Disables the generation of symbol table dumps.

Default Values:

At the start of compilation the default option settings are: `l+`, `d+`, `n+`, and `t-`. The default values are marked with an '*' in the descriptions above.

NOTE: When using the `l`, `e`, `h`, and `h` compile time options it is necessary to specify the size in single bytes. The compile-time options do not recognize 'K' bytes.

Compiler Listing Formatting Directives

The $PAGE, $TITLE, and $SUBTITLE directives can be included in Pascal source programs to create better organized titles and compiler listings. These directives must always start in the first column of the source line. The directive statements are not printed in the listing.

$PAGE

$PAGE causes a page eject in the compiler listing. When $PAGE is encountered, the rest of the line is ignored, and a page eject is queued. Note that a page eject is not performed immediately and will not be until a line of source text is encountered which must be listed. As a result, several $PAGE (also $TITLE and $SUBTITLE) requests can occur one after another, and only a single page eject will occur when a line of normal source text is finally found.

$TITLE and $SUBTITLE

The “$TITLE” directive specifies a program title line which is printed in the first line of the heading of every listing page. “$TITLE” may be followed by any text string to be used as a title. Title strings longer than 50 characters are truncated. “$SUBTITLE” is used to define a subtitle string and may have up to 60 characters. These directives can be used as often as desired to change the current page headings. Normally, a program will have a single title option at the very beginning of the source text, and subtitles will typically be used to name each major program section such as the global data declarations, each main procedure, etc.

A Sample Compilation Listing

As an aid in describing how to interpret the compilation listing, a small source program is shown below (Figure 1), and the corresponding compilation listing is shown on next page (Figure 2). The program is designed to demonstrate most of the features of a compilation listing and deliberately includes errors. Refer to these two examples for the discussion that follows.

Figure 2.1. Sample Source Program

```
$TITLE DumpReal
$SUBTITLE Global Definitions
PROGRAM dumpreal;
VAR
    badvar: ^anotherbadvar;
    i : integer;
    hexc: ARRAY[0..15] OF char;
    trix : RECORD
            CASE boolean OF
                true :(r: real);
                false:(c: ARRAY[1..5] OF char)
            END;
$subtitle Procedure PROCWITHERRORS
PROCEDURE procwitherrors;
BEGIN
    this demonstrates what error messages
    look like;
    END;
$Subtitle Procedure HEXVAL
PROCEDURE hexval(ch: char);
    BEGIN
    write(hexc[ord(ch) div 16], hexc[ord(ch) mod 16])
    END;
$SUBtitle M A I N L I N E
BEGIN
hexc:='0123456789ABCDEF';
WHILE true DO
    BEGIN
    write('Enter real number: '); prompt;
    readln(trix.r);
    FOR i:=1 to 5 DO
        BEGIN
        hexval(trix.c[i]);
        write(' ');
        END;
    writeln;
    END
END.
```

The program in figure 1 was compiled to produce the listing that follows using the command line:

```
PASCAL <DumpReal #20K
```

The command line initiates compilation of the program on a file called “DumpReal” giving 20K bytes of memory to the compiler.

Figure 2.2. Sample Compilation Listing

```
 Page   1  84/07/25 13:11:43  OS-9 Pascal - RS Version 02.00.00  DumpReal
 STMT  PLOC LEV     Global Definitions
-----+------+--+---------1---------2---------3---------4---------5---------6
    3     0D  0 PROGRAM dumpreal;
    3     0D  0 VAR
    5     0D  0    badvar: ^anotherbadvar;
    6    -2D  0    i : integer;
    7    -4D  0    hexc : ARRAY[1..15] OF char;
    8   -20D  0    trix : RECORD
    9   -20D  1           CASE boolean OF
   10   -20D  2              true :(r: real);
   11   -20D  2              false:(c: ARRAY[1..5] OF char)
   12   -20D  2           END;


 Page   2  84/07/25 13:11:43  OS-9 Pascal - RS Version 02.00.00  DumpReal
 STMT  PLOC LEV     Procedure PROCWITHERRORS
-----+------+--+---------1---------2---------3---------4---------5---------6
   14   -25D  0 PROCEDURE procwitherrors;
         ****   ^104,117
104: Identifier is not declared.
117: Unsatisfied forward reference.
*ERROR,  The following type-id's were assumed to be forward declared.
ANOTHERR
   15     0D  1 BEGIN
   16     3   2    this demonstrates what error messages
         ****         ^104         ^59  *SEE: 14
 59: Variable expected.
104: Identifier is not declared.
   17     8   2    look  like ;
         ****               ^51  *SEE: 16
 51: ':=' expected.
   18   11   2    END;


 Page   3  84/07/25 13:11:43  OS-9 Pascal - RS Version 02.00.00  DumpReal
 STMT  PLOC LEV    Procedure HEXVAL
-----+------+--+---------1---------2---------3---------4---------5---------6
   20     6   1 PROCEDURE hexvalfch: char);
   21     9   1    BEGIN
   22     3   2    write(hexc[(ord(ch) div 16)+1], hexc[(ord(ch) mod 16)+1])
   23    54   2    END;


 Page   4  84/07/25 13:11:43  OS-9 Pascal - RS Version 02.00.00  DumpReal
 STMT  PLOC LEV     M A I N L I N E
-----+------+--+---------1---------2---------3---------4---------5---------6
   25     6   1 BEGIN
   26     9   1 hexc:='0123456789ABCDEF':
```

```
   27    15   1 WHILE true DO
   28    23   2    BEGIN
   29    26   2    write('Enter real number: '); prompt;
   30    40   2    readln(trix.r);
   31    51   2    FOR i:=1 to 5 DO
   32    68   3       BEGIN
   33    71   3       hexval(trix.c[i]);
   34    90   3       write(' ');
   35    99   3       END;
   36   116   2    writeln;
   37   122   2    END
   38   125   2 END.


 Page   5  84/07/25 13:11:43  OS-9 Pascal - RS Version 02.00.00  DumpReal
 STMT  PLOC LEV     M A I N L I N E
-----+------+--+---------1---------2---------3---------4---------5---------6
PROC NAME       PSEC  PSIZE  LOCAL  STACK   CSEC  CSIZE
   0 DUMPREAL      3    130     27     15      4     37
   1 PROCWITH      1     12      0      7      2      0
   2 HEXVAL        2     55      0     13      3      0
                        197     27     35            37

38 Lines of source code compiled with 5 errors found, see: 17


 Actual Heap = 2661
Actual Stack = 3706
 Free Memory = 9581
```

Listing Page Headings

A three line page header is printed at the top of every listing page. The word “Page” is followed by a page number, which is followed by the current system date and time. The date and time is read only once at the beginning of a compilation, and this single time stamp is used for the heading of each page thereafter. To the right of the time stamp is the release level. Always be sure that the release level corresponds to the user manual or other reference documents that you may be using.

To the right of the release level is the program title string which was specified by a $TITLE directive in the program (see page 2-8). Note how the title string in figure 2 is derived from the title directive in figure 1.

The second line of figure 2 is typical of the form of every page heading. The column titled, 'STMT', refers to the sequence number of the corresponding source line. Note that the first statement number is 3, because line numbers 1 and 2 of the source text in figure 1 contain title and subtitle options respectively which are never themselves printed.

The next column titled 'PLOC' refers either to the variable (data) location counter or to the P-code (program) location counter, depending on whether the corresponding source line is a variable declaration statement or an executable program statement.

Source statement line number 6, for instance, shows that the data location counter is currently at -2 when the line is being compiled. The number is negative because storage is assigned backwards from the stack pointer. Since the line indicates the global variable '`i`' is being defined, and it is an integer which requires 2 bytes of storage, '`i`' will be assigned to location -4 in the global stack. Likewise, for line 7 of the program, the current data location counter is at -4 from the assignment of variable '`i`' on the preceding line. The variable '`hexc`' is being defined which is a 16 element character array. Since character type elements require 1 byte of storage, each '`hexc`' requires 16 bytes of storage total; thus, 'hexc' will be assigned to location -20. To find where the variable '`trix`' is assigned, look at the listing for source statement 14; '`trix`' is assigned to location -25. See Chapter 10 for more information on Pascal memory management.

Pcode is assembled beginning at location zero for every procedure. The 'write' call in line 22 then begins assembling at P-code location 0 for the procedure. For source line 23 you see that the termination code for the procedure begins at P-code location 42. The importance of showing the P-code location is for understanding the run time error messages. When an error occurs during the execution of a program, the procedure which caused the error and, if it is known, the P-code location within the procedure is reported. You can then easily find where within your program that the run time error actually occurred. Numbers under the 'PLOC' column, which contain data location counter values, are always followed by the letter 'D', and P-code location counter values are always followed by a space character.

The next column titled 'LEV' represents the compiler's 'lexical' level. The lexical level shows the nesting of control structures (loops, etc.) within the program and is helpful for finding unbalanced compound statements and declarations. Generally, those statements which have the same lexical level number are at the save level of nesting during the compilation scan of your program. Since the compiler always prints the image of a line of source program before it begins its scan of that line, the lexical numbers can sometimes lag behind what you might think they should actually be. Due to the number and complexity of conditions which cause the level number to be incremented and decremented, the best advice is to simply look at the numbers on your listings and gain a feel for how they move up and down.

A few spaces to the right of the title 'LEV', on the second header line, is the subtitle string. This is defined by the “$SUBTITLE” directive - see page 2-8.

The third line of each page heading is used to mark column boundaries as a visual aid. After the first 16 characters, which marks the required prefix of every line of source listing, is a scale showing the position of every 10th column of source text. If a parameter is used to define a short line width, then only that part of the scale which contains complete groups of 10 characters is shown. For example, if the line width is set to 72 characters, then the scale will show 7 groups of 10 characters.

Output Files Created by the Compiler

The Pascal compiler automatically creates two files during the compilation phase. They are:

PCODEF - the compiled P-code version of the Pascal program

PROCTAB - a compiler work file used to minimize memory requirements.

Unless command line parameters are used to specify alternate file names, the compiler will create files having these names in the current working data directory (usually the “PASCAL” directory or disk). You may delete the PROCTAB file any time after a compilation is completed if you wish, because it has no use outside of a compilation execution. When the compiler begins execution, it automatically deletes old PCODEF and PROCTAB files if they already exist. The deletion will normally not cause problems provided you either copy or rename any PCODEF files you want to keep after you have successfully compiled and tested a program.

Error Messages in Program Listings

There are two types of error messages. The first of which is shown in figure 2 after the listing of source line 14. The compiler displays a line beginning with four asterisks followed by an up arrow which indicates within a character position or two where the compiler was scanning when it realized that an error had occurred. Following the up arrow, the compiler prints the error number(s) for the error(s) found. On the following line(s) the compiler will print the full English error message text if it is able to open the file `PascalERRS`. This particular error also generated the second type of error message. Specifically, the message shown indicates that at this point in scanning the source program, the compiler realized that the identifier called 'ANOTHERB' (actually the identifier is

ANOTHERBADVAR, but the compiler recognizes only the first 8 characters of any identifier name) was previously declared to be a forward reference to an identifier which was not found by the time it was required to be found.

A more complete example of an error message is shown following the listing of source line 16. Here two errors were discovered for the source line, and the up arrows and error numbers appear as described above. Also, since this was not the first error message for the program, the text '*SEE: 14*' is added as a programmer aid. The 'SEE' tells you that the last error message previous to this current message occurred after the listing of source line 14. If you look at the end of figure 2, you can see a message indicating that 4 errors were found in the source program, and that the last message appeared after the listing of source line 17. The error message after source line 17 points you to the previous error message and that message points you to its preceding error message and so forth. You can quickly find all the error messages in your program without going through the whole listing. Error messages of the second type always occur with error messages of the first type, so you will be able to find all error messages by following the backward list.

Up to 9 error messages will be reported for any one line of source code. If more than 9 errors are found, a message is triggered that indicates that too many errors were found for the line. Sometimes one error will trigger other errors in the program. For example, if a variable declaration is being scanned and the reserved word 'PROCEDURE' is encountered, the compiler may get temporarily confused as to what it should really be scanning and multiple errors may be triggered as the compiler attempts to get on track again. Therefore, correcting one error may get rid of several other error messages.

If the listing of the source program is inhibited by compiler option(s), only source lines having errors and the corresponding error messages will be listed to let you perform “error checking” of long source programs without having to print the entire program.

The Compilation Statistics Report

The compilation statistics report is displayed after the last procedure. It shows memory usage by the compiler and is not related to memory requirements of the program compiled. The information in the report allows experienced programmers to optionally adjust the compiler's memory usage to decrease memory requirements or to permit the compiler to handle extremely large programs.

The Procedure Statistics Table

The procedure statistics table is always displayed after a compilation. It is not affected by the line width parameter option. The procedure table gives important information about the program and variable storage requirements of each procedure compiled and is helpful when interpreting any run time error messages. The procedure statistics table looks like the following example.

```
PROC NAME       PSEC  PSIZE  LOCAL  STACK   CSEC  CSIZE
   0 DUMPREAL      3    130     27     15      4     37
   1 PROCWITH      1     12      0      7      2      0
   2 HEXVAL        2     55      0     13      3      0
                        197     27     35           37
```

The information presented in each column is:

PROC is the procedure number assigned by the compiler. When runtime errors occur, they will be identified by this number. Procedure number zero always refers to the outer code block - the main program itself.

NAME is the first eight characters of the procedure name.

PSEC is the sector number within the P-code file where the P-code for each procedure begins. The PSEC value is used for compiler maintenance and is not of concern to the average programmer.

PSIZE gives the number of bytes of P-code instructions generated for the procedure. If a program is to be run with the PascalN interpreter, the total of the PSIZE column and the total of the CSIZE column (see below) given on the last line of the procedure table is the number of bytes of memory that will be required to store the program.

CSEC is the starting sector of the string constant block of the P-code file, CSEC is also used for compiler maintenance.

CSIZE is the length (in bytes) of string constants used by the procedure. A string constant is any sequence of 2 or more characters enclosed within two single quote marks.

LOCAL and STACK columns give the number of bytes that the compiler has calculated will be required for each procedure's variable storage (e.g., local and extended stack size). See Chapter 10 on Pascal Storage Management for more information on how this data is used.

Chapter 3. The P-code Interpreters

SYNOPSIS: Compiled Pascal programs in P-code form can be interpretively executed using either “PascalN”, the normal interpreter, or “PascalS”, a swapping (or “virtual memory”) interpreter which can run very large programs.

About The P-code Interpreters

After you have successfully compiled your Pascal program, you can run it using a “P-code” interpreter. It reads the P-code representation written by the compiler on the “PCODEF” (or other name you may have given) and executes the Pascal program. It also includes special features so that if an error occurs when the program is running, the type of error, the name of the offending procedure, and the corresponding source program statement line can be identified.

The OS-9 Pascal System includes two different P-code interpreters: PascalN which is the normal interpreter, and PascalS which is the swapping or “virtual memory” interpreter. PascalN is preferred for running most programs because it is fastest. PascalS is used only in cases where the compiled program is too large to fit in your computer's memory.

Calling the interpreter is simple: you just type its name followed by the name of the pcode file which is to be executed. For example:

```
pascaln pcode_filename
pascals pcode_filename
```

In the above example “pcode_filename” may be either the name of a pcode file in the current data directory or a pathlist to a pcode file in another directory.

A Detailed Description of P-code Interpreter Command Lines

The OS-9 command line used to call PascalN or PascalS can contain one or more of the following options:

1. Run time options
2. A parameter string to be passed to the Pascal program
3. Standard I/O redirection

The format for PascalN or PascalS command lines are:

OS9:PascalN Pcodefile [Run-Time Options] [: Params]

OS9:PascalS Pcodefile [Run-Time Options] [: Params]

Items enclosed in “[” and “]” are optional.

Standard I/O Paths

The three standard I/O paths associated with any OS-9 program are also used by the interpreters. The OS-9 paths for standard input, standard output, and standard error/status are associated with Pascal's built-in standard files called “INPUT”, “OUTPUT”, and “SYSERR”, These three files are always automatically opened by the P-code interpreter for use by your Pascal program. If the command line that called the interpreter did not redirect I/O, the default paths used are usually the user's terminal.

The standard file, INPUT, is opened for reading, the standard file, OUTPUT, is opened for writing, and the standard file, SYSERR, is opened in update (read and write) mode. These files can be used for interactive input/output or to access mass storage files. Some examples of calling the interpreters with I/O redirection are:

```
pascals  </d0/user5/indata s20k >/P

pascaln  </t2 >/t2 >>/t2 L10000   E4000

pascals  >/dl/reports/printfile   S10k E1000
```

For more information, see pages 4-8 and 8-2.

Run-Time Options

The interpreter command line can optionally include a list of run-time options which affect the interpreter performance and statistics reporting. They generally define memory allocation sizes, and/or they enable generation of the memory statistics report. The use of virtual code swapping buffers, which can be affected by these parameters, is discussed below. For more information about these options, refer to Chapter 11.

When using the PascalN and PascalS interpreters, the shell's “#memsize” directive has no affect on memory allocated. Memory allocation may be specified using one of two methods. The first is by using the compile time options to embed the requests in the code. This is particularly useful when making a program in Pcode form easy to run for users unfamiliar with all of Pascal's run-time options. The memory is allocated automatically, without user intervention, when initially requested through the compile time method. The second method is to use the run-time options to request memory from the command line.

There are a number of run-time memory allocation options which may be specified when using either of the interpreters to execute a program in pcode form. The options are used to control allocation of memory for the LOCAL STACK area, the EXTENDED STACK area, the initial HEAP STORAGE area, and when using PascalS, the SWAP BUFFER area. A single character is used to select one of the memory areas. The selecting character is then followed by an integer value representing the amount of memory you wish to allocate to that area. The integer values may be in one of two forms.

1. 0 to 65535 Byte sizes

2. 0K to 63K KByte sizes (1K equals 1024 bytes)

The selector character and the memory areas they set are shown below along with a brief description of their usage. For more detailed information on the individual memory areas see chapter 10.

“L” or “l”	LOCAL STACK
“E” or “e”	EXTENDED STACK
“H” or “h”	DYNAMIC HEAP
“S” or “s”	SWAP BUFFERS

Some examples of Run-time Options are:

```
PascalN pcodef L10000 E500
PascalN pcodef e300 h200 L400
PascalS pcodef L1000 S20K H1000
```

The Parameter String

The command line can optionally include a string of up to 80 characters, which will be made available to the running program in a special array called 'SYSPARAM'. The option can be used to pass to your Pascal program information such as initialization values, file names, etc.

NOTE: This is a feature of OS-9 Pascal and is not a standard Pascal function.

The array in which this data is passed to the program is predefined as:

```
sysparam : ARRAY [0..79] OF char;
```

The program can access sysparam as if it were specifically defined. The parameter string *must* begin with a ":" character (which is not passed to the program) followed by up to 80 characters. Extra characters are ignored. If less than 80 characters are given, or no parameter string is given, space characters will be supplied to fill out the array. The parameter string *cannot* include control characters or any of the characters that have special meaning to the OS-9 Shell (< > # ; or !). Here are examples:

```
pascaln pcodef :this message is passed to the program
pascaln sales.analysis  >/p :Report for Joanie's Boutique
pascals translate :channel5,channel8,noprint
```

Files Used by the Interpreters

As mentioned previously, the interpreter will read the Pascal program P-code from the filename you specify on the command line. The file "`SUPPORT`", which contains the machine language support library for virtually all parts of the OS-9 Pascal System, is required by PascalN and PascalS and will be automatically loaded from the current execution directory. You can also preload it using the OS-9 command "load support" to save time if you are constantly using the interpreters.

The file "`PASCALERRS`" should also be in the execution directory. `PASCALERRS` contains all of the run-time error message strings. If this file is not found in the execution directory, any run-time error will only print out the error number without the normal accompanying textual description. Also, if `PASCALERRS` is not in the execution directory, the standard procedure SYSREPORT (see the chapter on standard procedures) will not function correctly.

Choosing Between PascalS And PascalN

PascalN should be used to run P-code programs unless the combined size of the program itself and its data area is too large to fit in the memory available in your system. It runs faster than PascalS and is somewhat smaller.

PascalS allows you to run much bigger programs in the range of 10,000 plus source statements. In general, the ability of PascalS to run any Pascal program is mostly restricted by the amount of memory required for global and local variable storage. The ability to run very large programs is based on a technique called "virtual code swapping". The maximum possible size program that PascalS can run is 254 procedures within the outer block; each containing up to 32767 bytes of P-code (approximately 8.3 megabytes), however, this is practically limited by the amount of disk space available.

How Virtual Code Swapping Works

Despite its intimidating sounding name, the way virtual code swapping works is fairly easy to understand. It is also simple to use, because PascalS does it automatically with almost no special effort on your part. In fact, the main reason its internal operation is described here is so you can understand the factors that affect its efficiency and speed. Virtual code swapping is especially important because two major parts of the OS-9 Pascal System (the compiler and the native code translator) are also P-code programs run on PascalS.

All P-code programs, whether or not they are to be executed by PascalS, are automatically broken into 256-byte "virtual memory" pages by the compiler. PascalS copies these pages from the PCODEF (or equivalent) file to a temporary disk file. Of course, only one P-code instruction can be executed at a time, so theoretically only the page containing the current instruction must actually be present in "real" memory, and all other pages can remain on the disk until needed. The virtual memory effect is achieved because the disk file containing the virtual pages can be larger than the actual amount of real memory available.

Because disk read operations take a relatively long time, PascalS attempts to keep as many pages in real memory as possible in an area divided in 256-byte blocks called “swap buffers”. In order to further reduce the number of disk operations necessary, PascalS attempts to keep pages accessed most frequently in the swap buffers. If all, or nearly all, of the pages can fit in the swap buffers at the same time, PascalS can run almost as fast as PascalN. At the other extreme, if a relatively small percentage of the pages fit in the swap buffers, many more disk read operations will be needed so PascalS will run slower.

Putting all this together, it logically follows that the three main factors affecting the speed of PascalS ate:

1. The number of swap buffers - the more the better.

2. The ratio of the number of swap buffers to the total number of p-code pages.

3. The speed (access time) of the disk system.

There is always an absolute minimum of 8 swap buffers. Beyond that, the minimum number depends on the use of string constants in the Pascal program because procedures that use string constants have special pages allocated in the P-code file to hold the strings. As a procedure executes and makes reference to a string, the code page containing the string must be swapped into memory just like a normal code page. As long as that procedure is still active, that is, until the end of the procedure is reached and a return from procedure sequence is executed, any and all pages of string constants which were swapped in for the procedure must remain locked in memory. If a program were to use no strings at all, that program could be run using only the minimum 8 swap buffers.

There is an easy method for finding out if there are enough swap buffers available - simply run the program. If there aren't enough swap buffers, PASCALS will give a specific error message stating so. The “S” run-time option can be used to explicitly assign more swap buffers.

When allocating swap buffers, you will typically find that performance goes up in a lumpy fashion as more swap buffers are allocated, if, for instance, a program runs in, say, 5 minutes with 16 swap buffers, you might find that it runs in 4.5 minutes with 18 buffers, 4 minutes with 20 buffers, and 2 minutes with 21 buffers. In this case, a threshold is reached between 20 and 21 buffers. These thresholds are different for different programs and possibly for different executions of the same program. They depend on the procedure nesting structure, P-code sizes of various procedures, and which statements are executed in the program. For best performance, the rule is to simply allocate as much memory as possible for swap buffers.

Chapter 4. PascalT: The Native Code Translator

SYNOPSIS: The native code translator is an optional third step in the Pascal development process. The translator takes a compiled P-code program and converts it to 6809 assembly language source code ("native code"). The assembly language program generated is functionally identical to the P-code version of the program.

About the Native Code Translator

As mentioned in the introduction to this manual, an unusual characteristic of the OS-9 Pascal system is its ability to execute Pascal programs in P-code, native code, or both at the same time. "PascalT" is the program used to convert P-code versions of programs to native code (6809 machine language).

The *advantages* of using PascalT to translate P-code to native code are:

- native code is typically 5 times faster than the P-code version of the same program.
- native code is ROMable and reentrant (can be executed by two or more tasks at the same time).
- the memory space otherwise occupied by the P-code interpreter is available for other use.
- the assembler source code output can be edited by hand.

The *disadvantages* of native code programs are:

- native code is usually larger than the P-code version.
- virtual code swapping supported by PascalS is not available.
- the translation step takes additional time

Between one and three lines of assembly language source code will be generated for each byte of P-code translated. It works out to about 2.3 bytes of native code for each byte of P-code if string constants aren't counted. The extra memory can be offset somewhat by the availability of memory space otherwise occupied by the P-code interpreter. Since the compiler always reports the total bytes of P-code generated for each procedure and for the whole program at the end of every compilation, you can estimate how many lines of assembler source text and bytes of object code will be generated.

Since a large assembler source code output file may be generated, make sure the disk has enough free space to hold it before you begin translation. You may wish to create separate system and working diskettes containing only PascalT and the required files (`PascalT.PRUN`, `PascalT.MODL`, `PascalS`, `Support`, `PascalErrs`, and `Asm`, plus minimal OS-9 commands in the commands directory, and `Pascaldefs` in the working directory).

Running the Translator

PascalT is written in Pascal and is furnished in two parts:

PascalT.PRUN is the main translator program in P-code

PascalT.MODL contains the native code routines needed by the translator.

The "PRUN" file name suffix is a reminder that PascalT is a P-code file. Because PascalT is quite large, it must be run using PascalS, the virtual code swapping P-code interpreter. Here is an example OS-9 command line that starts the translator.

```
PASCALS pascalt.prun #15K
```

which calls PascalS to run the translator program, "pascalt.prun", using 15K bytes of memory for swap buffers. The size of the swap buffer memory given to PASCALS directly affects how fast PascalT runs. The memory given to PascalS should be made as large as your system will allow, preferably 15K or more. The auxiliary file “PascalT.MODL” is automatically loaded by PascalT without any action required by the user.

PascalT can be used in two ways: to translate a complete Pascal program, or to translate individual Pascal procedures one at a time. The second capability can be quite useful because Pascal applications can be developed that have some procedures in P-code and other (usually speed-critical) procedures in native code.

Run-time Environment

The run-time environment is described in detail in Chapter 10.

Communicating With PascalT

The first pact of running PascalT involves some dialogue with the user. When PascalT starts up, its first question will be:

```
Enter the name of the pcode file to be translated:
```

You should respond by entering the name of the P-code file you wish to translate followed by a ENTER. If you just respond with ENTER, PascalT will use the default file name, “PCODEF”.

PascalT first checks the P-code file to make sure that it was compiled without error and that it has not been altered since compilation. If the file does not pass these tests, one of the following error messages will be generated:

*Can't translate the pcode file because the file has K compile error(s).

*Can't translate the pcode file because the pcode file has been altered.

The next question you will be asked is:

```
Enter the name of the 6809 assembler language file to be produced:
```

The name you enter in response will be used as the file name of the text file created by PascalT to hold the assembler language output. If you only press the ENTER key, PascalT will use the default file name “PCODEFASM”. If the file specified already exists, it will be deleted, and a new file having the same name will be created.

Assuming all has gone well so far, the next question will be:

```
Translate all procedures? (Y or N):
```

You must respond with a single letter 'y' or 'n', in upper or lower case. You are being asked whether you wish to translate all procedures of a complete program, or if you wish to individually translate only certain individual procedures within the P-code file. A “y” response is the simplest, from the user's point of view, because all subsequent processing is completely automatic. An “n” response leads to a more complex and lengthy process that requires you to have a more in-depth understanding of the OS-9 Pascal System. A separate section of this chapter is devoted to each option.

Translating Complete Programs

If the response to the 'translate all' prompt, as described in the beginning of this chapter, is a 'y', the whole P-code file will be translated into native code.

The translator can't always determine exactly how much memory the native code program will require. The memory depends on how your program works. For example, a highly recursive program requires much more memory. Therefore, the translator arrives at an estimate, and gives you the opportunity to override the estimate if you so desire. The translator displays a table showing the local and extended stack size totals from the procedure table as well as any compile time overrides for local, extended, and minimum heap sizes. You will then be asked to supply values for the local, extended, and minimum heap allocations. To use the automatically calculated values, press ENTER in response to each question, or alternatively, enter a specific override value, which will be checked to ensure that it is within allowable limits.

Once the memory allocation questions have been answered, you will be prompted to:

```
Enter a name to be used as the module name:
```

Here you will enter the name that you want the final output tile to be called. If you press the ENTER key, the name of the program in the P-code file is taken from the first 8 characters of the name on the 'PROGRAM' line in the original source program.

You are also asked if you want line numbers in the P-code file to be translated with:

```
Translate line numbers in P-code file? (Y or N):
```

Inclusion of line numbers in the output file can only be accomplished if you included source line numbers in your P-code file via either a parameter or compile option as discussed in chapter 2 of this manual. If you respond “Y”, any included line numbers will produce native code to inform the run time error reporting system of the last known line number (see the chapter on run time error handling). If you respond 'n', then any included line numbers simply produce a comment in the assembly language source program.

Translating Individual Procedures

If the response to the 'translate all' prompt, as described in the beginning of this chapter, is a 'n' (no), the translator enters a mode that allows you to specifically name which procedures within the Pascal program are to be translated. This function is most commonly used:

- when you wish to create a program that mixes some P-code procedures and some native code procedures, or
- when you wish to translate individual “library” procedures of commonly used procedures for general use by P-code or native-code programs

One restriction is that the main program “outer block” cannot, be translated in this mode - you must translate it using the “translate all” option. Translating the outer block into native code requires many more considerations and much more processing than other procedures. However, it may call any number of external procedures which were translated individually.

The first prompt given in the individual translation mode is:

```
Produce external definition file? (Y or N):
```

The external definition file contains information about linkage requirements of each procedure and is used by the PascalE linkage editor in a later step when individual procedures are combined into a complete program. The usual response to this question is “Y” for yes. If you respond with 'n', you must manually create the external. definition file required by the PascalE utility program.

If you responded with “Y”, the next prompt will ask for the name of the external definition file to be produced with:

```
Enter the name of the external definition file to be produced.
```

If you simply hit the ENTER key, the default filename “PCODEINFO” will be used.

The next step is to give the translator a list of the procedures you wish translated. The procedure numbers are obtained from the procedure table given at the end of the compiler listing. The procedure numbers are requested by the prompt:

```
Enter a list of procedure numbers to translate.
A zero value will terminate the list.
A negative value will back out a previously selected entry.
```

You respond by entering a series of procedure numbers separated by spaces or carriage returns. You signify the end of the list of procedure numbers by entering the number “0” (remember that procedure 0 - the outer block - cannot be translated in this mode). If a procedure number was erroneously entered, simply enter the negative of the procedure number to remove it from the list. Each procedure number is checked to see that it really exists in the P-code file and, if not, the following error message is given:

```
Value N is out of range.
Absolute value must be between 1 and X inclusive.
Reenter.
```

where N is the bad procedure number and X is the highest allowed procedure number for the selected P-code file. If more than one procedure number was entered on a line, only those which were out of range are ignored, the others are accepted. If the selected procedure does not exist in P-code form, that is, if it was declared to be an external procedure, which means that it must already exist in native code form, the following error message is given:

```
Can't translate procedure N because it is an external routine.
```

where N is the bad procedure number. For either of these last two error messages,, you do not have to remove procedure 'n', because it was not accepted in the first place.

After the procedure number list has been completed, the list of accepted procedure numbers is displayed for verification, followed by the question:

```
Is this list correct? (Y or N):
```

If you respond with a “N” answer, you will be given an opportunity to correct the procedure list. Otherwise, the actual translation process will begin.

After the desired procedures are selected, source statements to produce a standard OS-9 memory module header are written to the assembler source file. The module header is given a hexadecimal value of 21 for the module type, indicating a machine language subroutine module (see the OS-9 system manual for a discussion of module formats and headers). The module data size is set to zero. For each procedure selected, a long branch instruction is inserted as the first code in the module. Each branch destination is to the startup code for a procedure. The branches are built in order of their actual procedure numbers, not in the order that they were selected.

If you requested an external definition file, you will be prompted for a pathname to be used for the external module name with:

```
Enter a pathlist to be used as the external module pathlist
and module name.
```

If the ENTER key is pressed, then the first 8 characters of the name on the 'PROGRAM' line in the original source program are used. Otherwise, the last name in the fully qualified path name is used as the module name, and the fully qualified path name is used to build the module path name in the external definition file. See the section on PascalE for more information on the the module path name.

If you are not having the translator build an external definitions file for you, the above prompt will be:

```
Enter a name to be used as the module name:
```

If the ENTER key is pressed, the first 8 characters of the name on the 'PROGRAM' line in the original source program are used. In both cases, the name is used as the module name in the assembly language source file being produced.

You are also asked if you want line numbers in the P-code file to be translated with:

```
Translate line numbers in P-code file? (Y or N):
```

Inclusion of line numbers in the output file can only be accomplished if you included source line numbers in your P-code file via either a parameter or compile option as discussed in chapter 2 of this manual. If you respond “Y”, any included line numbers will produce native code to inform the run time error reporting system of the last known line number (see the chapter on run time error handling). If you respond “N”, then any included line numbers simply produce a comment in the assembly language source program.

After one or more of the procedures have been translated and assembled, you will have to recompile the original source program to take advantage of the native code routines. The Pascal procedures that now have translated equivalents should be removed from the program and replaced with EXTERNAL procedure declarations. After the Pascal program has been recompiled, you need only update the P-code file with the PASCALE program, which puts into the P-code file all the information about the native code modules. The result is a program which uses both P-code and native code procedures.

Assembling Translated Procedures

The output from the translator is an assembly language source code text file, which must be assembled using the OS-9 assembler. Procedures are translated one by one, with optimization being performed as each line of assembler source code is produced. There are many reduction rules used to optimize the native code procedures for both memory and speed efficiency. The resulting file can then be assembled by the standard OS-9 assembler. The file `PASCALDEFS` must be in the data directory during the assembly process. You can, if needed, edit the assembler source file with a text editor to further optimize the code, but you must be very careful to retain the overall structure and function of the module. The source code must always adhere to the complete memory and register usage conventions as described in the chapter on writing procedures in native code. An example of calling the assembler (using the default assembler output file name) is:

```
ASM PCODEFASM #20k
```

Of course, any of the assembler options can be selected as well, especially the “`L`” option to produce a listing and the “`O=`” option to override the default output file name. In general, it is a good practice to make sure that the output filename is the same as the module name. The “`O`” option or “`O=`” option must be specified if you wish to create an executable output file from the assembler. Using the assembler without either form of “`O`” option is useful to make sure the assembly is error free before actually assembling to create an executable file.

The output of the assembler in the above example is a native code module a file in the execution (“CMDS”) directory that is also called “PCODEFASM”. If the `PCODEFASM` file contained only external modules from a partially translated program, they are now ready to be linked to the recompiled P-code program using PascalE. If the `PCODEFASM` file contained a fully translated program, it may now be run. See the section below on Running Native Code Programs.

Pascal programs that are translated to native code do have runtime error checking, but when errors occur it is not easy to determine at what line of which procedure the error occurred as reported by the P-code interpreters. Also, the P-code interpreter run time options “`l`”, “`e`”, and “`s`” have no meaning when running native code programs. You should only translate procedures or programs into native code after you are sure that they are fully functional and have been completely tested using one of the interpreters.

Running Native Code Programs

This section describes how to execute a Pascal program which has been totally translated to native code using PascalT. These programs are executed directly from the OS-9 Shell. The Shell command line format to run a native code Pascal program is:

```
program <input >output >>syserr [Run-Time Options] [:Params]
```

Items enclosed in “[” and “]” are optional.

Where:

“program” is the pathname of the native code program which is to be executed. The module must have been created using PascalT and the OS-9 Assembler as described previously. It may reside in the system current execution directory, or it can be preloaded into memory using the OS-9 “load” command.

“input” is the pathname to be associated with the standard file 'INPUT'. The file will be opened for read access only, and it must already exist when the program begins. If the path name is given, it must be immediately preceded by the “<” character. If the input path is not redirected, any program reference to the standard file INPUT will default to the standard input path, usually the terminal.

“output” is the pathname to be associated with the standard file 'OUTPUT'. The file will be opened for read and write (i.e. update) access, and, if the file does not already exist, it will be created. If the pathname is given, it must be immediately preceded by the “>” character. If the output path is not redirected, any reference to the standard file OUTPUT will default to the standard output path, usually the terminal.

“syserr” is the pathname to be associated with the standard file 'SYSERR', a predefined text file provided as an OS-9 Pascal extension. The file will be opened for read and write (i.e. update) access, and, if the file does not already exist, it will be created. If the pathname is given, it must be immediately preceded by the “>>” characters. If the syserr path is not redirected, any program reference to the standard file SYSERR will default to the standard error path, usually the terminal.

“Run-Time Options” is a list of zero or more run time options which generally define memory allocation sizes, and/or they enable generation of the memory statistics report. The only two options which have any affect for programs which run totally as native code are the '`h`' and '`i`' options. See Chapter 11 for details.

“Params” is a character string of up to 80 characters which will be made available to the running program in a predefined array called 'SYSPARAM'. This works in a manner compatible with the P-code interpreters.

The variable stack space size of native code programs is estimated by PascalT and the size value is put into the program's memory module header. The OS-9 Shell “`#`” memory size option can optionally be used to override the default size. Highly recursive programs, or programs that use large dynamic arrays, should be given additional memory using this option.

Native code programs that have EXTERNAL declarations assume that the procedures involved are separate memory modules, so PascalT generates code to automatically perform the OS-9 “link” and “load” functions in order to access them.

The “Support” run-time package module must either be preloaded in memory or must be present in the system's current execution directory in order to run any native code program. The `PascalErrs` file must also be present if full text error messages are desired.

Chapter 5. “PascalE” - The Pascal Linkage Editor

SYNOPSIS: “PascalE” is a linkage editor used to create a single program consisting of a “mainline” program having EXTERNAL declarations and one or more separately compiled external native code procedures. PascalE is only used when separate compilation is desired or when P-code and native code procedures are mixed in the same program. PascalE is intended for use by advanced programmers.

An Overview of PascalE

Whenever a Pascal program contains one or more EXTERNAL routines, the P-code file for that program must be edited using PascalE after compilation before it can be executed. PascalE inserts addresses of external native code routines in the appropriate places in the program so the external procedures can be called.

PascalE is an interactive program. For each external procedure, you must supply the file name of the module containing the native code version of the procedure, the offset from the starting execution address of the module for the first instruction of the procedure, the type of the module, and the local and extended stack requirements of the external procedure. Normally, all this information is built automatically by PascalT, in which case the operation of PascalE is almost trivial. If the external modules were not created by PascalT (such as when the external module is hand-written in assembly language), you need to know how external procedures are handled by OS-9 Pascal so you can properly interface them.

When PascalN or PascalS begins processing a P-code file, one of the first actions taken is to check for the existence of external procedure declarations. When one or more of such declarations are found, a special table in the P-code file is consulted for more information about each declaration. First, the interpreter needs to know if the module containing the native code external procedure is already loaded into memory. If it is not, it needs to know the path name of the module so that it can be loaded into memory. After ensuring that all required modules are in memory, the interpreter needs only to complete its internal tables for each procedure with its execution address and its local and extended stack sizes. It may be that all external routines are contained in a single module, but it is likely that some of the required external routines are in other modules. As an example, a program which does signal processing may call a native code procedure to do Fourier analysis on the data, and also call several routines in a different module to do digital plotting of the results.

Running PascalE

To run the PascalE utility, use the command form:

```
PASCALE <extdeffile :pcodefile
```

“extdeffile” is the name of the file containing the external definition file records to be processed. This file is automatically generated by PascalT using the default file name “PCODEFINFO”. If you are attempting to link hand written native code, you can omit the path name and the preceding “<” character if you want to enter the definition file records from the standard input path (usually the terminal). In this case, simply enter one or more “use” records (which are discussed further on) from the terminal to cause the proper external definition files to be processed. If input is from a terminal, remember that “use” records begin with a space character, and that after all definition file records have been entered you need to key in the "end of file" sequence (usually the escape or ESC key, which is CLEAR+BREAK on the color computer).

“pcodefile” is the path name of the P-code file to be updated with the external file information. The file name must be immediately preceded by a colon character — no leading space characters or other characters are permitted. There is no default name assumed, so that if you wanted, for instance, to refer to the default P-code file produced by the OS-9 Pascal compiler, you would enter ":PCODEF".

If the ":"character is omitted, or if it is not followed immediately by a path name, the following error message will be given:

```
*ERROR No P-code file name has been found. A P-code file name must
be supplied immediately following a ":" on the command line.
```

If the P-code file doesn't exist, or if there is an OS-9 error encountered while trying to open the file, the following error message is given:

```
*ERROR OS-9 error x encountered when trying to open file.
```

where "*x*" is the OS-9 error number. Refer to the *OS-9 Commands Manual* for the meaning of the error number. If the P-code file is found to have been altered, which could produce catastrophic results, or if the file is not a P-code file after all, the following error message is given:

```
*ERROR P-code file has been altered or has invalid format.
```

If there were compile time errors found when the P-code file was produced, the. P-code file is unusable, and the following error message is given:

```
*ERROR P-code file has x compile time errors, it cannot be processed.
```

"*x*" is the number of compile time errors which were detected. If the P-code file has no EXTERNAL procedures declared within, the following error: message is given:

```
*ERROR P-code file has no external routines.
```

If a record is found within the external definition file which has a space character in the first character position, but is not immediately followed by the letters "use", another space character, and then a path name, the following error message is given:

```
*ERROR Invalid command on line number x at USE file nesting level of y
```

Where "*x*" is the current line number in the text file which is being processed, and "y" is the use file nesting level as described earlier in this section. If a valid "use" command is encountered, but the referenced file cannot be opened, the following error message is given:

```
*WARNING OS-9 error x encountered while trying to open USE file y
```

Where "*x*" is the OS-9 error number, and "y" is the path name of the file from the "use" record. If a record in the external definition file is not a valid comment, use, or definition record, the following error message is given:

```
*ERROR Invalid EXTERNAL definition format on line number x at USE
file nesting level of y
```

Where "*x*" is the current line number in the text file which is being processed, and "y" is the use file nesting level as described earlier in this section. The previous error message is also given if any field of a definition record is not in a valid format — such as a number being out of range, or a name being too long or being composed of the wrong type of characters.

After PascalE has completely processed the external definition file, it checks to see if all declared external procedures have been mapped. If not, a list is produced of all of the procedure names, procedure numbers, and module types which were not processed during this execution. It is possible that you processed these procedures during a previous execution or that you will process these procedures in a future execution. You must, however, cause every external procedure to be mapped by PascalE before you will be allowed to run or even translate the P-code file.

Finally, if you are going to run the P-code file via one of the OS-9 Pascal interpreters, you can use PascalE between executions to remap the external procedures. If you translate a complete P-code file into a native code program, however, the mapping that exists at the time of the translation is permanently encoded into the native code program - to change that mapping you must re-execute PascalE for the P-code file, and then retranslate the P-code file. Partial translations of a P-code file are not permanently affected in this way.

The External Definition File

If you want to link hand written external assembly language procedures to a P-code program, you must understand the format of “external definition files”. There are three types of records that can exist in the external definition file.

Comment Records

Comment records start with an “*” character in the first character position followed by any other printable characters that you desire. The line is ignored by PascalE and is allowed simply to give you a way to put descriptive information into the file. You should always insert comment records to name the original source program from which the external definition records were derived, the version of that program (if such information is kept), and a time and date stamp of when either the external definition file was created or when the original source program was compiled. If you intend that others might use your external routines, you should also include some sort of documentation of the use of the routines — at least a simple statement naming each procedure and its use.

Use Records

The second type of record is the “use” record. It consists of a single space character followed by the letters “USE”, followed by another space character, followed by a path name; upper and lower case letters are considered to be equivalent. The use of “use” records allows you to construct nested or subordinate structures of external definition files. The path name given refers, of course, to the name of another text file containing further external definition records. This other file could even contain its own “use” records. The level of nesting is restricted only by the amount of extended stack space available when PascalE is run. As each “use” record is encountered, processing of the current external definition file is temporarily suspended while the referenced file is processed.

When PascalE first begins execution, the “USE file nesting level” is zero. When the first “use” record is found, the nesting level goes up to 1. If the referenced file also has a “use” record, the nesting level will go up to 2 while the newly referenced file is being processed. Naturally, the nesting level goes back down by one after each referenced file is processed. The nesting level is important only for interpreting any error messages which might be produced by PascalE. After the last record in a "use" file is processed, processing continues with the record after the "use" record which referenced that last file.

Definition Records

The third type of record is the “definition” record. Each record has exactly 6 fields of information, and can be up to 127 characters in length. The first field is the name of the procedure being described as it would be named by a source program. The name must be between 1 and 8 characters in length. The first character must be a letter, but succeeding characters can be letter, digit, or underscore characters. Immediately after the procedure name there must be a comma, and after that a module name. The module name can be up to 116 characters in length and gives the path name of the module containing the native code version of the defined procedure. Normally, the module name field contains a fully qualified path name so that any programs which use the external procedure can be run regardless of any setting of data and execution directories. The module name can be any valid OS-9 path name.

When PascalN or PascalS processes an external procedure table entry, it first scans the module name starting at the end and working toward the beginning until either the beginning of the module name is found, or until a slash character is found. The name retrieved is used for an attempted “link” operation, and, if the link fails, the whole module name is used for an attempted “load” operation. Refer to the *OS-9 Technical Information* manual for a description of link and load operations. The link and load is also qualified by a “module type” field which is described next.

The module name must be immediately followed by a comma, and then a module type. A module type is a 1 to 3 digit number that must be in the range from 0 through 255. External procedure modules produced by PascalT will always have a type of 33 which indicates a native code subroutine module. The module type is used to qualify the link or load operation on the module. Following the module type there must be a comma, and then an offset number. The number must be in the range of 0 through 32767, and when added to the execution address of the linked or loaded module gives the address of the first instruction to be executed for the procedure. If there is more than one procedure in a module, the normal convention is to place a three byte “long branch” instruction for each procedure as the first executable code for the module. Thus the first procedure in the module would have an offset of 0, the second an offset of 3, and so forth.

Following the offset field is a comma, a local stack size number, another comma, and an extended stack size number. Both of these numbers must be in the range of 0 through 32767 and must conform to the true requirements of the procedure as described in various chapters of this manual. To recap, a “definition” record looks like:

```
proc-name,module-name,type,offset,local-size,extended-size
```

When PascalE begins execution, one of the first things that it does is to make a table of all of the procedure names which were declared as EXTERNAL within the source program. As the external definition file is processed, the first external definition record that matches each procedure name in the table is used, all duplicate matches then being ignored. The implication is as follows. A particular program may have, say, 17 procedures declared as external. They are to be mapped to three different modules. One external procedure, however, has the name of “CLEARSCR” which is the name of a procedure in two of the three modules. It's quite possible that only one of the modules has the correct CLEARSCR procedure which is needed in this instance, the other module's CLEARSCR might perform some totally undesirable action — it just so happens that both modules just ended up having a procedure with the same name. To make sure that PascalE maps your external CLEARSCR references to the right module, make sure that the correct external definition record is processed first. That way the second, or any subsequent CLEARSCR definition records will be ignored. You could, of course, create a text file containing only the exact required definition records, but it is more usual to have a single definition file for each module and to simply use "use" records to cause them to be processed in the correct order.

A problem that is more or less the opposite of that just described is also possible. You may want to reference both CLEARSCR procedures in your source program. Clearly you cannot do so by declaring that both procedures are external with the same name. The solution, however, is easy. In the source program, declare two different procedures as external with different names. You can still call one CLEARSCR if you wish, or maybe you can call one CLEARSC1 and the other CLEARSC2. Now you just need to create an external definition file for the module which contains the procedure referenced as say, CLEARSC1. The easiest way to do this is to make a copy of the original module's external definition file, and use the OS-9 text editor program to change the name of the procedure from CLEARSCR to CLEARSC1. Similarly, create an external definition file for the other module and change the name on the appropriate definition record from CLEARSCR to, say, CLEARSC2. Now use these two new external definition files when you run PascalE against the P-code file for your source program.

There is one thing that you must be very careful about when using external procedures. There is no way to ensure that the calling code in the source program correctly matches the way that the external procedure needs to be called. For example, an external procedure may be a Pascal function that returns a real number and takes an integer and a character array as an argument. Your source program, however, could declare the external procedure to be a Pascal procedure which takes 4 real numbers and 5 integers as arguments. OS-9 Pascal has no way to verify that any program is correctly referencing an external procedure. The program will compile correctly and PascalE will correctly map the actual call of the external procedure to the right native code instructions. When the program actually executes, however, any such mismatched reference is likely to destroy the integrity of the running system, quite likely with catastrophic results! A word of advice then — make sure your source program calls really match up with what the external routine expects. If the external module was produced by PascalT, then as long as you refer to that module only from the source program which was used to create that module, and as long as you don't change any external procedure's type (i.e. Pascal function or procedure) or its parameter list, then you need not worry.

Chapter 6. A Pascal Language Summary

SYNOPSIS: This section is a summary of the rules used to construct Pascal programs. As mentioned in the Introduction, this manual is not intended to serve as a comprehensive language reference, so the main purpose of this chapter is to provide an overview of ISO standard Pascal plus characteristics and features of OS-9 Pascal that are system dependent. See Appendix B for a detailed Backus-Naur Form description of the language.

Alternate Character Sets

The standard alternate character set is supported by OS-9 Pascal. This allows use of comments, arrays, and pointers on systems with limited character sets. The following table shows the allowable alternate characters:

Standard	Alternate	Usage
{	(*	Begin comment
}	*)	Close comment
[	(.	Begin array range/element
]	.)	Close array range/element
^	@	Pointer character

Notation Used in Descriptions

In order to give concise descriptions, this manual uses a simple notation system to show how parts of Pascal are used. The rules of this system are:

- Pascal keywords are capitalized.
- Things that the programmer specifies (such as expressions, variable names, definitions, etc). are shown in lower-case.
- Things that may optionally appear *once* are enclosed by brackets [ ] .
- Things that may optionally appear *one or more* times are enclosed by braces { }

Source Program Format

The Pascal compiler uses source files which can be prepared using any text editor. Input lines can be up to 127 characters long, but only the first 110 characters are processed. Each input line is terminated by a carriage return (ENTER) character. No distinction is made between upper case and lower case letters except within string constants.

Identifier Names

Identifiers are names given to Pascal variables, types, procedure and function names, etc. Identifiers can be of any length, but only the first eight characters are significant, i.e./ used for matching and to determine uniqueness. They include any combination of the following characters:

A..Z	a..z	0..9	_

Note: Use of the underscore character is a nonstandard OS-9 Pascal extension.

The first character of an identifier must be an upper or lower case letter (A..Z or a..z) and upper and lower case letters always match. Here are some examples of Pascal identifiers:

```
x
```

```
sum
COUNT
Part20

Last_Year     matches   last_year
COUNTER_200   matches   COUNTER_400  (first 8 chars, match)

5Days         ILLEGAL - can't begin with digit
_temp         ILLEGAL - can't begin with “_”
```

General Program Organization

A Pascal program starts with the “PROGRAM” statement followed by various optional declarations, then program statements. A general outline of program organization is shown below.

Example 6.1. General Pascal Program Organization

```
PROGRAM name(files)

LABEL  <label declarations>
CONST  <constant declarations>
TYPE   <type declarations>
VAR    <global variable declarations>

PROCEDURE and/or FUNCTION subprograms

BEGIN
  <main program statement>
END.
```

The main program is called the “outer block”, because it is executed first when the Pascal program is run. In simple programs, all the work may be done by the main program statements in the outer block, however, very frequently much of the work is done by subprograms called “procedures” and “functions”. These subprograms can contain their own internal ("local") variable, type, label, and similar declarations, and have a similar organization as the outer block including the possibility of having additional subprogram definitions within themselves. All declarations and names declared within subprograms are only “known” within that subprogram or additional subprograms it may contain.

The PROGRAM Statement

The PROGRAM statement must be the first statement of the program. It states the program name and may optionally include names of files (I/O paths) used within the program. Unlike some other Pascal compilers, OS-9 Pascal does not require that files be declared in the PROGRAM statement, however, if you do so they will be checked for correctness. Some examples:

```
PROGRAM test
```

```
PROGRAM sort(infile,outfile,scratchfile)
```

Comments and Blank Statements

Comments consist of text which begin with either a “{” or a “(*” character sequence and end with either a “}” or a “*)” character sequence. They are ignored by the compiler and may appear anywhere in a program. Comments may also extend over multiple lines. For example:

```
{ This is a comment }

(* This is also a comment *)
{ So
  is
  this } { This will also be a comment *)
```

Blank lines may also be used freely to improve program readability.

Indentation of Lexical Levels

It is traditional and considered good programming style (but is not mandatory) to indent statements in Pascal programs so that “lexical levels” of control structures (loops, compound statements, etc.) are clearly visible. Indenting makes the program logic much easier to read and understand. The compiler assists you by printing the lexical level of each statement on the compiler listing.

LABEL Declarations and GOTO Statements

“Labels” are used to indicate places in the program GOTO statements transfer control to. Frequent use of GOTO statements is unnecessary in Pascal because of the wide variety of structured control statements. Excessive use of GOTOs can result in programs that are hard to read, test, and maintain. Use of GOTOs can be justified in a few special cases, such as error handling.

Each line number used in the program must be declared in the LABEL statement before it is used. Each label must be a unique number in the range of 0 to 9999. GOTO statements cannot transfer control to a statement outside the current procedure. Here is an example of its use:

```
PROGRAM label_demo;
LABEL 10,20;
VAR x,temp:INTEGER;
BEGIN
  ioabort(input,false);
  FOR x := 1 TO 100
    DO BEGIN
      readln(temp);
      IF ioresult(input) <> 0 THEN GOTO 10;  {end if input error}
      writeln('The square root of',temp,' is ',sqrt(temp));
    END;
  GOTO 20;
10: writeln('input error!!!');
20: END.
```

IOABORT and IORESULT will be discussed in Chapter 8.

CONSTANT Declarations and Constants

The CONST statement allows a name to be given to a constant value. Although constant numbers and strings may be used in the program without prior declaration, the CONST facility is useful for defining numbers that may have to be changed in future or different versions of the program, and also increase program readability.

The CONST keyword can be followed by any number of definitions of numeric or string constants. Each definition has the form:

<identifier> = *<constant>* ;

Numeric constants may be preceded by “+” or “-” signs. The constant definitions end when another Pascal keyword is encountered. Here is an example program segment illustrating use:

```
CONST Pi = 3.1415926;
HighLimit = 22.5;
Title = 'Test Result Summary';
EndLine = $0D;
```

Numeric Constants

Numeric constants can be of integer or real types. Any numeric constant can contain an underscore after the first character. Integers are whole numbers in the range -32768 to +32767. OS-9 Pascal will also accept integer constants in hexadecimal notation using a “$” followed by one to four hexadecimal digits (0-9, A-F). Real numbers are numbers that include decimal points and/or the exponential “E” notation. Real numbers have a range of approximately 2.9*10^39 to 1.7*10^38 with about 9 1/2 significant digits. Here are some examples of numeric constants.

```
    -5            legal integer constant -5
    9999          legal integer constant 9999
    9_999         legal integer constant 9999
    $1DF          legal integer constant - hexadecimal 1DF
    $1df          legal integer constant - hexadecimal 1DF

    _612          ILLEGAL - can't start with “_”
    120000        ILLEGAL integer - too big
    $1C49A2       ILLEGAL integer - too big

    1.45          legal real constant   1.45
    .0303         legal real constant   .0303
    15E44         legal real constant   15*10^44
    1.234e-12     legal real constant   1.234 * 10^-12
    1_000_000.    legal real constant   1000000.

    _100_000      ILLEGAL - can't start with “_”
    5._56         ILLEGAL - can't start with “_” even after “.”
```

String Constants

String constants are sequences of up to 100 characters enclosed within single quote characters. Here are some examples:

```
    'X'
    '5'               note: character “5” NOT numeric value 5!
    'abcdefg'
    'strings can have embedded spaces'
    'PAGE 10'
```

The single quote character used to enclose strings may itself be contained within the string if two are used. Each pair of adjacent single quotes has the literal value of one single quote character. Some examples:

`'Joe''s Grill'` has the actual value of: Joe's Grill

`'AABB''''DDEE'` has the actual value of: AABB''DDEE

String constants are considered to be of type ARRAY[1..N] of char. Type checking is performed when making assignments to variables. If a string constant is assigned to an array with a lower bound other than 1, an error will be reported.

Type and Variable Declarations

The TYPE and VAR statements are used to describe and declare variable storage, respectively. One of Pascal's outstanding features is its support for variables, arrays, sets, files, and record structures which can be composed on a number of basic “built-in” data types or user-defined data types. The subject of variable declarations in Pascal is quite complex and beyond the scope of this manual, however, the information that follows describes characteristics of OS-9 Pascal that are specific to its particular implementation of the language.

Hierarchy of OS-9 Pascal Data Types.

Pointers

Structured Types:
- ARRAY
- RECORD
- FILE
- SET

Simple types:
- REAL
- Ordinal Types (subranges permitted):
  - CHAR
  - INTEGER
  - BOOLEAN
  - Enumerated (user-specified)

The amount of actual memory space required for each simple variable, array element, or record element of the simple types are: CHAR and BOOLEAN, one byte; INTEGER, two bytes; REAL, five bytes.

Arrays:

Arrays may be multi-dimensional and can be of any simple or user defined type. A new type definition can be included in the array declaration. Index values must be of an ordinal type, OS-9 Pascal permits the character sequences “(. .)” to be used as array subscripts in addition to the standard brackets “[ ]”.

Sets:

May have up to 256 members each. OS-9 Pascal has extended the syntax for constructing set constants. If the constant is to contain a series of successive elements, they can be specified using the form 'low..high' as in:

```
workday := [Sunday, Tuesday..Friday];
```

PACKED Structures:

Because the 6809 microprocessor uses byte addressed memory, all arrays, structures, etc., are inherently “packed” (except bit-level BOOLEAN). The PACKED attribute keyword is properly scanned for correctness by the compiler but has no actual effect on the program.

FILE Types:

Each variable declared as a FILE type is associated with a 16-byte control block which is automatically initialized when the code block making the definition is executed, however, the file itself is not automatically opened (except for the standard I/O paths). When the block of code is exited, all files are automatically closed. FILE types cannot be passed by value in subprogram calls.

Procedure And Function Declarations

Procedures and functions may be declared within the main program, or within other functions or procedures. Declarations are almost identical to the PROGRAM declaration except for the keywords used (PROCEDURE or FUNCTION), and the ability to include a “formal parameter list” which is used to receive or return data to calling procedures. Otherwise, the body of a function or procedure is the same as the main program and may include “local” declarations for constants, labels, variables, etc, as well as executable statements.

Procedure General Format.

```
PROCEDURE name(parameter-list);
LABEL label declarations
CONST constant declarations
TYPE type declarations
VAR local variable declarations
PROCEDURE or FUNCTION subprogram declarations
BEGIN
  procedure-statement
END.
```

Function General Format.

```
FUNCTION  name(parameter-list) : result-type;
LABEL label declarations
CONST constant declarations
TYPE type declarations
VAR local variable declarations
PROCEDURE or FUNCTION subprogram declarations
BEGIN
  function-statement
  {Note assignment to function name in body of function}
  name := result-type-value
END.
```

PROCEDURES are called as statements in a program to perform some operation. FUNCTIONS are called from within expressions and return some value. The format of the parameter list declaration is similar to a VAR declaration and can specify a number of simple or complex variables of any type.

Pascal includes a large number of “standard functions” which are built-in functions that perform commonly used operations such as evaluation sines, cosines, exponentials, etc. Standard functions are also comprise a large part of Pascal's input and output capabilities.

OS-9 Pascal recognizes two special subprogram declarations: FORWARD and EXTERNAL. FORWARD is used to tell the compiler about the existence of a procedure in advance (e.g., before it is actually declared). FORWARD may be required if the procedure(s) involved are referenced in procedures declared ahead of the procedure. For example:

```
PROGRAM forward_example;
  (declarations)
PROCEDURE second(c,d:INTEGER); FORWARD;
PROCEDURE first;
  BEGIN
    a := 1;
    b := 2;
    second (a,b);           (call to FORWARD procedure)
  END;
PROCEDURE second            (Second now declared param. list
  BEGIN                     is NOT repeated)
    writeln(c);
    writeln(d);
  END;
BEGIN
  first;
END.
```

The EXTERNAL procedure directive tells Pascal that the procedure declared is for an external, assembly language procedure created by the native code translator (PascalT) or hand-written in 6809 assembly language. After compilation of the program, the PascalE editor is used to link all external procedures to the main program. For more information see Chapter 5. Here is an example:

```
PROGRAM main;
  VAR a,b,c: INTEGER;
  (variable declarations)
  PROCEDURE my_extern(x,y,z:INTEGER); EXTERNAL;
  BEGIN
    (statements)
    my_extern(a,b,c);            (call to external)
  END.
```

Function and procedure calls in OS-9 Pascal exactly follow the ISO standard with the exception of the following minor differences:

1. PROCEDURE and FUNCTION types cannot be passed by name to other procedures or functions.
2. FILE types cannot by passed by value.
3. The EXTERNAL procedure directive is *not* defined in the ISO specification, however, it is commonly found in sophisticated Pascal compilers that permit separate compilation of subprograms or calls to assembly language subprograms.

Assignment Statements and Expressions

The simple assignment statement evaluates an expression and assigns the result to a variable. For example:

```
result := temp+5;

x := sum MOD base;

val := (total/count)+ adj

answer := (last >first) AND NOT correct
```

In general, the variable in which the result is stored may be a simple type, a specific element of an array, a function, or a specific element of a record structure. The type of the variable and the type of the expression result must match (except integers and reals), or an error will be reported by the compiler. Automatic type conversion is performed when real and integer types are mixed within an expression.

Expressions are made up of “operands” (data) and “operators”. Operands may be constants, variables, array or record elements, set members, pointers, etc. Numeric operands may have their sign specified by use of the + or - single argument operators (which are different than add and subtract operators).

The operators are evaluated in precedence (priority) order. If several operators of the same priority appear consecutively, they are evaluated from left to right. The normal precedence may be overridden by grouping of subexpressions in parentheses. The operators and data types that may be used are given in the table on the next page in order of *group* precedence.

A number of standard functions that perform mathematical and logical functions are also available - see Chapter 7.

Pascal Operators

Operator	**Function**	**Argument Type**	**Result Type**
ARITHMETIC OPERATORS			
+ (unary)	Identity	integer, real	same as operand
- (unary)	Sign inversion	integer, real	same as operand
*	Multiplication	integer, real	int, real (Note 1)
/	Division	integer, real	real
&	bit-by-bit AND	integer	integer (Note 7)
DIV	Int. division	integer	integer (Note 2)
MOD	Modulo	integer	integer (Note 6)
+	Addition	integer, real	int, real (Note 1)
-	Subtraction	integer, real	int, real (Note 1)
!	bit-by-bit OR	integer	integer (Note 7)
#	bit-by-bit XOR	integer	integer (Note 7)
BOOLEAN OPERATORS			
NOT	logical negate	boolean	boolean
OR	logical “or”	boolean	boolean
AND	logical “and”	boolean	boolean
RELATIONAL OPERATORS			
=	Equivalence	Note 3	boolean
<>	Not equal	Note 3	boolean
<	Less Than	any simple, str.	boolean
>	Greater Than	any simple, str.	boolean
<=	Less or Equal	Note 4	boolean
>=	Greater or Equ.	Note 4	boolean
IN	Membership	Note 5	boolean
SET OPERATORS			
*	Intersection	set	same as operands
+	Set Union	set	same as operands
-	Set Difference	set	same as operands

Note 1: The result is integer if both operands are integers, otherwise the result is real.

Note 2: The result is truncated division (remainder is discarded).

Note 3: Arguments can be any simple, string, pointer, or canonical set type.

Note 4: Arguments can be any simple, string, or set type.

Note 5: The left operand must be ordinal type T; the right operand must be set type.

Note 6: The MOD statement differs between the Wirth/Jensen and the ISO specifications. The ISOMOD standard procedure can be used to specify the desired method. (See Chapter 7)

Note 7: Bit-by-bit logical operators are non-standard extensions to OS-9 Pascal which are not included in the ISO specification.

Extensions to the Assignment Statement

OS-9 Pascal includes enhancements to the assignment statement that are not part of the ISO standard. These extensions permit “implied loops” to be incorporated into single assignment statements that involve assignments of sets or one-dimensional arrays (vectors).

Set constants can be built using the subrange form “A..B” where “A” and “B” are constant members of the set being formed. The form denotes a list of elements from A through B inclusive. The ordinal position of A in the set must be before the ordinal position of B. An example is:

```
PROGRAM setdemo;
TYPE
   days = (sunday, monday, tuesday, wednesday,
           thursday, friday, saturday);
VAR
   workday,
   weekend: SET OF days;
BEGIN
weekend := [sunday, saturday];
workday := [monday..friday];
END.
```

One dimensional arrays (vectors) with character elements can be indexed using the “expression FOR constant” form. Where “expression” yields an integer “first-index” value and “constant” yields an integer number-of-elements value. If the character array is multidimensional, this form can only be used to index the last dimension. For example:

```
PROGRAM chardemo;
VAR
   a,b: ARRAY [1..26] OF CHAR;
   i,j : integer;
BEGIN
a:='abcdefghijklmnopqrstuvwxyz';
b[1 FOR 26] := a[1 FOR 26]; {copy array a to array b}
END.
```

Compound Statements

Groups of statements may be collected together and syntactically treated as a single statement. The statements are enclosed with BEGIN and END as shown below:

```
BEGIN
  x := y*44.5;
  z := x/2.0;
END;
```

The main body of a Pascal program or subprogram is technically defined as a single “statement”, so a compound statement is used to enclose the program statements. In this case, the closing END is followed by a period instead of a semicolon. OS-9 Pascal does not require the final END statement - if it is missing when the end of the source file is reached, END will be assumed.

The body of Pascal loops are also syntactically single “statements”, so BEGIN and END are used to include multiple statements in the body of the loop. For example:

```
WHILE y < 12 DO
  BEGIN
    t := t+d[y];
    y := y+i;
  END;
```

Looping and Conditional Statements

The IF-THEN-ELSE Statement

The IF-THEN statement is Pascal's basic decision making statement. The decision is based on the result of a boolean expression; if the result is true, the statement(s) following THEN are executed, otherwise the statements after the ELSE statements are executed. The ELSE part is optional. For example:

```
IF x < Pi
  THEN writeln('x is smaller than pi');
(next statement)

IF x < PI
  THEN writeln('x  is smaller than pi')
  ELSE writeln('x is larger or equal to pi');
(next statement)
```

Compound statements can be used within IF-THEN statements, and multiple IF-THEN statements can be nested. Some examples:

```
IF count < required
  THEN writeln('there are too few units')
  ELSE IF count > too_many
    THEN BEGIN
      hit_max := TRUE;
      writeln("there are too many units');
    END
    ELSE WRITELN('there are the correct number of units');
```

The CASE statement

The CASE statement uses the value of an expression (that gives an ordinal result) to select one of a numbered list of statements to execute. For example:

```
CASE val OF
  1: writeln('value is one');
  2: writeln('value is two');
  3: writeln('value is three');
  4: writeln('value is four'};
END;
```

OS-9 Pascal has extended the syntax for case statements in two very useful ways. First, an 'OTHERWISE' selection is provided. If the 'OTHERWISE' selection is not used and a case statement is executed with the value of the selecting variable not appearing in the selection list, a run time case error is caused, and the program is aborted. The 'OTHERWISE' option allows a convenient method for specifying “don't care” or catch-all processing within a case statement. Also for a single case selection, if multiple successive values are part of the same selection, you can use the form 'low..high' as part of the specification. The next example demonstrates the use of both of these enhancements.

```
CASE nextchar OF
   'a'..'z'   : processletter;
   '0'..'9'   : processdigit;
   '$',' '   : processspecial;
   OTHERWISE : processother;
   END;
```

Constants in the constant list of a case statement can have the form “A..B” which designates the list of values from A through B inclusive. The ordinal value of A must be less than the ordinal value of B. For example:

```
PROGRAM casedemo;
TYPE
   days = (sunday, monday, tuesday, wednesday,
           thursday, friday, saturday);
VAR
   s : days;
BEGIN
s := monday;
CASE s OF
   monday..friday : writeln(' WEEK DAY ');
   OTHERWISE      : writeln('  WEEKEND ');
   END { CASE }
END.
```

The REPEAT Statement

The REPEAT statement creates a loop where the exit test is performed at the end of the loop. The test expression must yield a boolean type result. The body of the loop is always executed once. The REPEAT statement is different than other loops in that is does not require a BEGIN.. end compound statement because the UNTIL keyword serves to define the end of the loop. The general form for REPEAT statements are:

```
REPEAT
  (statements)
UNTIL boolean-expression;
```

For example:

```
count :=0;
REPEAT
  writeln('input a number');
  readln(num);
  sum :=sum+num;
  count := count+1;
UNTIL count = 10;
writeln('the average value is",sum/10);
```

The WHILE-DO Statement

The WHILE-DO statement creates a loop where the exit test is performed at the beginning of the loop. The test expression must yield a boolean type result. If the first test is false, the loop body is never executed. If more than one statement is to be used in the body, a compound statement must be used. The general form for WHILE-DO statements is

```
WHILE boolean-expression DO
  (statement)
```

For example:

```
done := FALSE;
count :=0;
WHILE NOT done
  DO BEGIN
    readln(data[count]);
    IF data[count] = 0 THEN  done := TRUE;
    count:=count+1;
  END;
```

The FOR Statement

This statement creates a loop that uses a specified variable to automatically count iterations of the loop. The two forms of this statement are:

```
FOR variable := start-expr TO end-expr DO statement;

FOR variable := start-expr DOWNTO end-expr DO statement;
```

The control variable must be an ordinal type (CHAR, INTEGER, etc.) thus, REAL variables cannot be used. The variable must be a simple variable (not an array, structure element, or parameter) declared in the declaration part of the same program or subprogram in which it is used. It can be used within expressions but cannot be altered by program statements in the body of the loop.

When the FOR statement is first executed, the starting and ending expressions are evaluated. Both must be of a compatible type as the counting variable. The variable is set to the result of the starting expression and the loop body is executed. Each time through the loop the counting variable is increased by one (or decreased by one if the DOWNTO form was used). The loop terminates when the counting variable reaches the value of the end-expression.

An example:

```
PROGRAM fibonacci;
{ print first ten numbers of the fibonacci series }
VAR counter,sum: INTEGER;
BEGIN
 sum := 0;
 FOR counter := 1 TO 10
   DO BEGIN
     sum := sum+counter; {'counter' used but not changed }
     writeln(sum);
   END;
 END.
```

The WITH-DO Statement

This statement provides a convenient way for different elements of the same record to be accessed in a single statement. The syntax of the WITH statement is:

```
WITH <record identifier> DO
   <statement>
```

Within the statement (which may, of course, be a compound statement), the field names of the record structure are given without the record name. The record name is supplied by the compiler from the name immediately following the WITH keyword. This statement not only saves typing, but it allows the compiler to produce better compiled code by reducing the number of times record addresses must be computed. Here is an example of its use:

```
TYPE
  person =
    RECORD
      name : ARRAY [1..25] OF CHAR;
      age : INTEGER;
      can_vote : BOOLEAN;
    END;
VAR
  index : INTEGER;
  people : ARRAY[1..3] OF person;

BEGIN
  people[1].name := 'Alan Jones               ';
  people[2].name := 'Betty Baker              ';
  people[3].name := 'David Miller             ';
  people[1].age := 14;
  people[2].age := 35;
  people[3].age := 17;

FOR index := 1 to 3 DO
  WITH people[index] DO
    BEGIN
      can_vote := age >= 18;
      IF can_vote THEN writeln(name,' is old enough to vote')
    END;
END.
```

Chapter 7. Standard Functions and Procedures

OS-9 Pascal includes a library of standard functions and procedures that perform a variety of useful tasks. These subprograms are “built-in” OS-9 Pascal and can be used in programs without any special declarations. This chapter describes the subprograms for mathematical and run-time control. Input/Output related standard subprograms are discussed in Chapter 8.

OS-9 Pascal includes all ISO standard functions (except DISPOSE, PACK, AND UNPACK) plus a number of additional non-standard functions, which are denoted by an asterisk where they appear in the descriptions that follow.

Standard Procedures:

FIELDPUT
ISOMOD
MARK
RELEASE
NEW
MATHABORT
RIGHTJUST
SYSTIME

Standard Functions:

ABS
ADDRESS
AFRAC
AINT
ARCTAN
CHR
CNVTREAL
COS
EXP
FIELDGET
LN
MATHRESULT
ODD
ORD
PRED
ROUND
SHELL
SIN
SIZEOF
SQR
SQRT
SUCC
TRUNC

Standard Procedures

The following is a list of non-I/O-related standard procedures available. I/O related procedures are discussed in Chapter 8, The expected arguments and their type is given in the title line, followed by a description of the procedure's operation. Those procedures which are extensions added to OS-9 Pascal, but not called for in the ISO standard, are marked by “*”. Programs that use these functions may not be portable to other Pascal compilers.

Procedure FIELDPUT(variable-name, start-bit, length, value:integer)*

This procedure allows you to selectively set or reset bit fields in an integer. The start-bit defines the integer starting bit number of the field to be affected; 0 is the least significant bit and 15 is the most significant bit. Length defines the integer number of bits in the field, 1 through 16. Bits are affected starting at start-bit and proceeding toward the least significant bit with wrap around through the high order bit positions if necessary. The value will be stored in the specified field within the 16 bit 'integer-variable-name' variable. Only the least significant “length” bits of the value are used, any higher order bits are ignored.

Procedure ISOMOD(logical-value:boolean)*

If the value supplied is TRUE, use of the MOD operator works as per the language specification (this is the default mode). A value of FALSE implements the more classical MOD which allows a negative right argument and can return a negative result.

Procedure MARK(variable-name:pointer-type)*
Procedure RELEASE(variable-name:pointer-type)*

MARK sets the current top of heap pointer to the value of the pointer variable. RELEASE is the opposite: it resets the current top of heap pointer to the value of the pointer variable.

Procedure NEW(variable-name:pointer-type)

Creates an undefined ("global") variable of the pointer's type which is referenced by the pointer. Storage for the new variable remains allocated even if allocated within a subprogram which is later exited.

Procedure MATHABORT(logical-value:boolean)*

If the value supplied is TRUE, arithmetic overflow, range error, and divide by zero errors will cause the program to abort (this is the default mode)* A value of FALSE will not abort the program and will keep the least significant bits of the result for integer overflow conditions and return zero for divide by zero conditions(See 12-4).

Procedure RIGHTJUST(logical-value:boolean)*

If the value supplied is TRUE, then strings and character arrays are right justified as per the language specification for WRITE and WRITELN calls (this is the default mode)* A value of FALSE yields left justification. This affects formatting of character fields when a field width specifier is given.

Procedure SYSTIME(year, month, day, hour, minute, second:integer)*

Returns the current system date and time in the integer variables. If the system clock is not active or has not been initialized, zeros are returned.

Standard Functions

The following is a list of non-I/O-related standard functions available. I/O-related functions are discussed in Chapter 8, The expected arguments and their type, and the type of the value returned by the function is given in the title line, followed by a description of the function's operation. Those functions which are extensions added to OS-9 Pascal but not called for in the ISO standard are marked by “*”. Programs that use these functions may not be portable to other Pascal compilers.

Function ABS(expression:integer-or-real): same-type-as argument

Returns the absolute value of the argument, e.g., ABS(x) = |x|.

Function ADDRESS(variable-reference): integer*

Returns the actual memory address of the the specified variable. The variable reference can be a simple or complex type, or an element of a complex type. Note: actual memory addresses of program variables can change each time the program is run.

Function AFRAC (expression:real) : real*

Returns the fractional portion of the value of 'real-expression'. For instance, the result of AFRAC(3.14) is 0.14, and the result of AFRAC(-1.5895E-3) is -0.0015895.

Function AINT(expression:real): real*

Returns the integer portion of the value of 'real-expression'. For instance, the result of AINT(3.14) is 3.0, and the result of AINT(-0.0123) is 0.0.

Function ARCTAN(expression:integer-or-real): real

Performs the inverse trigonometric arc tangent function cosine on an integer or real argument expressed in radians. The result is of type real.

Function CHR(expression:integer): char

Returns a char-type result having the same numerical value as the integer argument, e.i., convert integer to char.

Function CNVTREAL(string-or-char-array): real*

The string constant or the contents of the character array is converted to a real value. If a string constant is used, it must be at least 2 characters long. If a character array is used, a carriage return must be put in the array to terminate the conversion process. A carriage return character has a decimal value of 13.

Function COS(expression:integer-or-real): real

Returns the trigonometric cosine of an integer or real argument. The result is of type real and expressed in radians.

Function EXP(expression:integer-or-real): real

Performs the exponential function to argument, e.g. `e` to the X power. The argument may be of types integer or real.

Function FIELDGET(expression, start-bit, length:integer): integer*

This function allows you to selectively extract bit fields from an integer valued expression. The start-bit defines the integer starting bit number of the field to be extracted; 0 is the least significant bit and 15 is the most significant bit. Length defines the integer number of bits in the field, 1 through 16. Bits are extracted starting at start-bit and proceding toward the least significant bit with wrap around through the high order bit positions if necessary. The extracted field is then right justified into a 16-bit result with any unused high order bits set to zero.

Function LN(expression:integer-or-real): real

Computes the natural logarithm (base `e`) of an integer or real type argument. The result is of type real.

Function MATHRESULT: integer*

Returns the last error number detected during any math operation since the last call to MATHRESULT. The error numbers can be retrieved only if the MATHABORT flag is false, either by a MATHABORT(FALSE) call or by a the runtime '`A-`' option. See the section on runtime options and the MATHABORT procedure for wore information. If no math errors have been detected, then the result of the MATHRESULT call will be zero. The error numbers which can be returned are shown in the chapter on error messages.

Function ODD(expression:integer): boolean

Returns boolean value of TRUE if the value is an odd number. Argument must be of type integer.

Function ORD(ordinal-type-value) : integer

Takes an argument of any ordinal type (char, integer, boolean, etc.) and returns the ordinal number of the value within that type. For example ORD(Wednesday) of the set (Monday..Friday) is 2.

Function PRED(ordinal-type-value): same-type-as-argument

Returns the predecessor value that has an ordinal value of one less than the argument. For example, PRED(7) returns 6; PRED(FRIDAY) returns THURSDAY. If no predecessor exists, (such as PRED(0)) an error occurs.

Function ROUND(expression:real): integer

Rounds the real argument up or down to the nearest whole number and returns the result as a type integer value. The result must be small enough to be represented as an integer or an error occurs. Some examples: ROUND(12.1) returns 12; ROUND(12.9) returns 13; ROUND(100000.4) returns an error because the result is too large.

Function SHELL(string-or-char-array): integer*

This function calls the OS-9 SHELL, passing the string or character array in the parameter area to be executed as an OS-9 command. SHELL can be used to access any OS-9 command, utility, or to run other concurrent or sequential programs. The error code returned from SHELL is converted to an integer by prefixing 8 bits of zeros. If a string constant is used, it must be at least 2 characters long. If a character array is used, a carriage return character must be put in the array to terminate the command to SHELL. A carriage return character has a decimal value of 13.

Function SIN(expression:integer-or-real): real

Returns the trigonometric sine of an integer or real argument. The result is of type real and expressed in radians.

Function SIZEOF(variable-or-type-name): integer*

Returns the size (in bytes) of a simple variable, a type, or a complete data structure. Cannot be used on individual elements of arrays or structures.

Function SQR(expression:integer-or-real): same-type-as-argument

Computes the square (X*X) of a real or integer argument. Returns a real or integer result.

Function SQRT(expression:integer-or-real): real

Computes the square root of a positive real or integer argument. Returns a real result.

Function SUCC(ordinal-type-value): same-type-as-argument

Returns the successor value that has an ordinal value of one more than the argument. For example, SUCC(7) returns 8; SUCC(THURSDAY) returns FRIDAY. If no successor exists (such as SUCC(TRUE)) an error occurs.

Function TRUNC(expression:real): integer

Truncates (removes any fractional part of) a real argument and returns a integer-type result. For example TRUNC(12.75) returns 12 as an integer type. The truncated argument must be small enough to be represented or an error occurs.

Chapter 8. Input/Output Functions and Procedures

Pascal I/O operations are based on use of standard procedures and functions which are part of the standard library. Since any given computer can have a variety of different input and output hardware devices, the built-in I/O functions must be non-specific and the ISO standard reflects this. Because the OS-9 operating system allows access to all I/O files and devices using essentially the same calling methods, Pascal programs can read or write any system I/O device or file.

Pascal's I/O handling facilities (or lack thereof) and the way they work are commonly cited as a significant flaw in an otherwise superbly designed language. In order to overcome these traditional limitations, the OS-9 Pascal standard library includes highly intelligent I/O functions and an extensive variety of additional (and non-standard) functions so that Pascal programs can fully utilize the computer's I/O facilities including interactive and random-access I/O.

Files are used to perform all I/O to terminals, printers, disk files, etc., and in general, standard I/O functions will work with any I/O device. This type of operation works well with the device-independent design of the OS-9 operating system, i.e., a Pascal “file” is analogous to an OS-9 “path”.

Pascal recognizes two categories of file objects: TEXT files ("textfiles") and the structured FILE type. Textfiles are what their name implies: text in the form of characters organized into lines terminated by an end-of-line character. The basic and most commonly used I/O functions operate on textfiles. Both TEXT variables and FILE-type variables are declared in the program or subprogram variable declaration section (except predefined files discussed on the next page).

In the standard I/O procedure descriptions that are given in the following pages, each procedure's name is followed by a list of expected parameters. You can usually tell which class of I/O the function applies to by the description of the first identifier in the list as follows:

“text-filename” means the function must be used with textfiles

“file-variable” means the function must be used with structured file types

“filename” means the function can be used with either of the above types

Procedures that use one or more parameters called “external-filenames” directly accept a standard OS-9 file name. The OS-9 filename must be passed as a string constant or a character array. File names stored in a character array must be terminated by a character which OS-9 will recognize as the end of a valid path name such as a space or carriage return (ENTER) character.

Predefined Standard I/O Files

Three file names, INPUT, OUTPUT, and SYSERR, are predefined as textfiles. They cannot be redefined in a program. Output records can be any length, and input records can be up to 127 characters long. These three files correspond to the three OS-9 standard I/O paths, and may be redirected when the program is run by means of the OS-9 Shell “<” “>” and “>>” operators.

Name	OS-9 Path #	Attributes
INPUT	0	Input only (user keyboard)
OUTPUT	1	Input or output (user display)
SYSERR	2	Input or output (user display)

Note

The “SYSERR” predefined file is a non-standard extension to OS-9 Pascal.

Differences Between Interactive And Mass-storage Files

There are important differences between the processing of files which are directed to interactive (SCF) type files such as terminals, versus mass-storage (RBF) type files. When WRITEs are issued to an SCF-type device, the OS-9 Pascal support system will remove any trailing blanks from the line. This is usually highly desirable for reducing the amount of time it takes to write a line to a terminal, and is the way you would expect I/O to occur. A similar function occurs when a fixed length file is directed to an SCF device and a GET is issued. In this case, after the carriage return character (ENTER) is given, the remainder of the fixed length record is padded with space characters, enabling you to declare a file to be a fixed length array of characters and to read records in a “user friendly” manner. The GETCHAR function permits single, unedited input characters to be read. The IOREADY function can be used to test if a character has been typed and is waiting to be read. These two functions are invaluable for writing menu-driven screen-oriented programs.

The biggest difference between the processing for RBF versus SCF type devices lies in the fact that buffers are not loaded until needed for SCF type files. For RBF type files, I/O is handled as described in the language specification - the buffer for a file is loaded with the contents of the first record as soon as the file is opened and EOLN and EOF status is available immediately. For SCF type files, since records are not loaded into the buffer until they are needed, the EOLN and EOF status of the file may be slightly different then expected. The reason for the differences in the handling of the two types of file devices is to provide a more “natural” type of file handling. For example, if the buffer for an SCF type device was loaded as soon as the file was opened, it would be difficult to issue a prompt to the terminal to tell the user what was being expected for input - the user would have to type in his first input line before he could see what it was he was supposed to be entering.

Fortunately there are only two major considerations that you need to be aware of to properly handle the difference in processing between the two device types. First, you need to know the differences concerning EOLN and EOF signaling. Since, when the file is first opened for an SCF type device, the buffer is not preloaded, it is not possible for either EOLN or EOF to be set. The system can't tell, of course, if the user is going to hit the carriage return (ENTER) or end of file keys until you actually read the first record, which is not true for RBF type devices since the first record is pre-loaded as soon as the file is opened, and the support system can easily determine if either the EOLN or EOF condition exist for the first record in the file, A similar problem occurs whenever a READLN call is made against an SCF type file. READLN will clear out the current buffer and set EOLN to false, but again, the next record is not actually read until it is needed, so EOLN and possibly EOF cannot be detected until after the next record is actually read. When READLN is issued against an RBF type file the current buffer is cleared and the next record is immediately loaded from the disk file - thus EOLN and EOF is immediately known for the next record.

The second consideration is simply a programming method which eliminates most of the processing problems which arise from the differences in handling between the two file types. Whenever a new line is to be processed simply code the statement:

```
IF interactive(filename) THEN get(filename);
```

which simply says that if the file “filename” is an interactive (i.e. SCF type) device, then get the next record. If “filename” is an RBF type device, the buffers are automatically preloaded as needed, and the statement will not do the “GET” operation. If, however, the file is an SCF type device, issuing the “GET” call will cause the buffer to be immediately loaded so that EOLN and EOF will be correctly set. Remember, only issue this call when the next line is to be processed, like immediately after the file is opened and the first line is needed, or after EOLN has been detected for a record and READLN has been issued to clear the current buffer.

Standard I/O Procedures

Procedure APPEND(text-filename {,external-filename {,open-mode}})*

APPEND identical to RESET described below except that it can only be applied to variable length record (i.e. text) files. Only PUTs and WRITEs will be allowed, and new records are automatically appended to the end of the current text file. Records currently in the file are not destroyed as with REWRITE. For a description of “open-mode”, see the RESET/REWRITE Procedure description.

Procedure CLOSE(filename)*

CLOSE performs an explicit close of the designated file. All files except the three predefined files are always implicitly closed upon exit of the block in which they were declared. No error is generated if the file is already closed.

Procedure GET(file-variable)
Procedure PUT(file-variable)

GET advances the file pointer to the next component of the file (i.e., read a record from the file). PUT appends the value of the file variable to the file (i.e. adds a record to the file). If the end of file is reached upon a GET, EOF is set to true and the value of the data read is undefined. Although the ISO specification says that the file pointer must be at the end-of-file for PUT to work, OS-9 Pascal allows records to be read or written in any order (see the REPOSITION standard procedure).

Procedure GETINFO(filename, 32-byte-structure)*
Procedure PUTINFO(filename, 32-byte-structure)*

These procedures read or write a copy of the 32 byte option area in the OS-9 path descriptor that defines line input editing and control character definitions. The second argument must be the name of an identifier which has a size of 32 bytes. Use these calls with great caution as their misuse could produce strange results. Refer to the *OS-9 Technical Information* manual for a description of the option area.

Procedure IOABORT(filename, logical-value)*

If the logical value is FALSE, nonfatal I/O errors associated with the filename will not abort program execution. After each I/O call you should use the IORESULT function to check results until an IOABORT call is again given with a TRUE value. If an I/O error occurs for a file which has disabled its I/O abort flag and a previous error number has not been cleared via a call to IORESULT, only the previous error information is kept, the new error is number is lost (See 12-4).

Procedure OVERPRINT(text-filename)*

Outputs a carriage return without a linefeed.

Procedure PAGE(text-filename)

PAGE is similar to WRITELN except that a page eject (form feed) is generated, so output appears on the next physical page. Operation is somewhat hardware dependent in that the output device must recognize the form feed character.

Procedure PROMPT(text-filename)*

PROMPT causes the current text buffer to be immediately forced out without either carriage return or linefeed characters.

Procedure READ({file-variable,} read-parameter-list)
Procedure WRITE({file-variable,} write-parameter-list)

These procedures permit one or more variables to be read from or written to a file without the need to manipulate the file variable for each item to be read or written. As with the primitives GET and PUT, records may be accessed in sequential or random order. The parameter list must be a list of variable names (READ), or a list of variable names, expressions, and formatting directives (WRITE).

NOTE: A special nonstandard characteristic of WRITE using formatted output is the case of the “:w:d” format directive for REAL numbers. If “d” is zero, the number is printed in “integer” format without a decimal point and trailing zeros.

Procedure READLN({text-filename,} read-parameter-list)
Procedure WRITELN({text-filename,} write-parameter-list]

These procedures permit one or more variables to be read from or written to a (sequential) textfile. A parameter list identifies the name(s) of the variable(s) to be read or written. The parameter list must be a list of variable names (for READLN), or a list of variable names, expressions, and formatting directives (WRITELN).

NOTE: A special nonstandard characteristic of WRITELN using formatted output is the case of the “:w:d” format directive for REAL numbers. If “d” is zero the number is printed in “integer” format without a decimal point and trailing zeros.

Procedure REPOSITION(file-variable, record-number)*

Repositions the file pointer to a specific fixed length record within the specified file. To update records in random order, for each record perform the following steps: REPOSITION to the record, GET the record, REPOSITION back to the record, PUT the record. The record-number parameter can be either an integer or real value expression or variable name.

Procedure RESET(file-variable {,external-filename {,open-mode}})
Procedure REWRITE(file-variable {,external-filename {,open-mode}})

RESET repositions the file pointer to the beginning of the file so it can be reread from its beginning, e.g., the file is rewound. REWRITE is similar except all data is erased (and the file becomes empty) so it may be rewritten.

An optional second argument (which must be a string constant or an ARRAY OF CHAR) can be used to specify an external OS-9 filename. If the second argument is not given, the external file name defaults to the identifier used to name the file in the program. If RESET or REWRITE is used against a file that is already open, the action depends on whether or not a second argument is given; if no argument is supplied, the file is rewound and the file is marked as input or output only. If the second argument is given, the current file is closed and the file name given is opened and marked as input or output only. If the file referenced in a REWRITE call already exists, it will be effectively deleted before it is opened.

An optional third argument (which must be an integer expression) is used to determine the attributes for creation of the file and the access mode for the file. When a file is created, the 8 low order bits are used to determine the file's attribute byte. The 8 high order bits are used to determine the file's access mode when the file is initially opened. If this argument is missing a default of $0303 will be assumed (user Read/Write file attribute byte, Read/Write permitted to the file) . For additional information on attribute and access byte contents, see sections on I$Create and I $Open system calls in the *OS-9 Technical Information* manual.

Procedure SEEKEOF(file-variable)*

SEEKEOF sets up the file control block so that the next write or put operation will add records to the end of the file. A READ or GET would return an end of file condition.

Procedure SHORTIO(file-variable, record-length)*

SHORTIO sets the record length in the file control block to the value of the integer variable or expression 'record-length'. The next PUT/GET operation can then write/read a short record. After the PUT/GET operation is performed the record length in the control block for the file will be reset to the length declared for the file in the source program. SHORTIO is usually used after a GET is performed on the file and a 'short record' error is returned. By executing the following statements:

SHORTIO(filex, LINELENGTH(filex)); PUT(filex);

after the file has been correctly positioned (see REPOSITION, described above) you can rewrite the short record back to the file without getting record length errors and without extending the length of the short record. You cannot set the record length to a negative number or to a number larger than the declared record length with this procedure. See Chapter 9 for more detail on SHORTIO.

Procedure SYSREPORT({text-filename,} integer-value)*

SYSREPORT is used for automatic standard error reporting. It searches within the file `PascalErrs` (which must be in the current execution directory) for the record (error message) number passed in the integer parameters. The first record of the file is record 1 for this procedure only. The text of the error message is then written to the referenced text file. If the text file name is not given, along with the following comma, the standard file “SYSERR” is assumed. This routine allows you to append your own messages to the `PascalErrs` file and have an easy method of generating messages. The text of the message is not automatically forced out to the text file, you will generally have to complete the error reporting sequence with a PROMPT(text-filename) or WRITELN(text-filename) call. This allows you to further append information to the error message with WRITE calls, if necessary.

Procedure UPDATE(file-variable {,external-filename {,open-mode}})*

UPDATE is identical to RESET and REWRITE described above except that it can't be applied to variable record length (i.e. text) files, and the buffer is not loaded with the first record when the file is first opened. Both GETs and PUTs will be allowed as long as a series of only GETs or a series of only PUTs is done. If you want to do one or more GETs followed by one or more PUTs, or vice-versa, you need to call REPOSITION in between the two types of calls. Any records currently in the file are not destroyed as with REWRITE. For a description of “open-mode” see the RESET/REWRITE Procedure description.

Procedure WRITEEOF(file-variable)*

WRITEEOF sets the file's EOF mark to the current position in the file. Note that issuing a REWRITE to a file automatically marks the file as empty. WRITEEOF allows marking any valid record number in a fixed length record file as the last record in the file. Any records which were previously written beyond the current record position are lost.

Standard I/O Functions

Function EOF{(filename)}: boolean

EOF returns TRUE if the file current position is at end-of-file, FALSE otherwise.

Function EOLN{(filename)}: boolean

EOLN returns TRUE if the current file position points to an end-of-line character in the file's buffer. EOLN can only be used with textfiles.

Function FILESIZE(file-variable): real*

FILESIZE returns the number of whole records currently in a fixed length record file. FILESIZE does not provide an indication if the last record in a file is only a partial record due to improper file creation. Any attempt to GET a partial record will cause an error.

Function GETCHAR(filename): char*

GETCHAR returns a single binary byte from the input buffer if available, otherwise waits for a character to be entered. GETCHAR is very useful for single keystroke menu selected, reading nonstandard I/O devices, etc.

Function INTERACTIVE(text-filename): boolean*

INTERACTIVE returns TRUE if the file is an interactive (i.e. SCF) device, FALSE otherwise. There are several differences in the way that I/O is handled for interactive devices. You may want to code your program to properly handle both cases. Interactive input files do not pre-load the file buffer when the file is opened.

Function IOREADY(filename): boolean*

Returns TRUE if there is at least one character in the OS-9 input buffer; FALSE otherwise. Useful for detecting if a key has been pressed on an interactive device. A subsequent input function must be used to actually read the data.

Function IORESULT(filename): integer*

Returns the error code result of the last operation performed on the file (see IOABORT above). If no error occurred since the last IORESULT call, zero is returned. Error numbers less than 100 are Pascal errors while others are OS-9 errors. Each call to IORESULT returns the current error number and then resets the current error number to zero. If any I/O operation needs to return an error number and finds that an error number has been already saved (i.e. not cleared by a call to IORESULT), the previously saved number is kept and the new error number is lost.

Function LINELENGTH(filename): integer*

Returns the length of the record currently in the buffer. This is the length of the last record read or the current length of the record which is about to be written.

Function OPENED(filename): boolean*

Returns TRUE if the file is currently opened, FALSE otherwise.

Function POSITION(file-variable): real*

Returns the record number of the next record to be read or written (i.e. the current file position pointer).

Chapter 9. Suggestions for Program Optimization

SYNOPSIS: This section contains information and suggestions on how to produce more efficient programs and how to use special features of OS-9 Pascal.

The Debug Option

Use the debug option (see the chapter on compile-time options for `d+/d-` option) during program development to aid in producing bug free programs. The debug option causes the compiler to create code to perform run time range checking on memory references via pointers, and assignments to boolean, set, and subrange type variables. Once individual procedures or whole programs have been thoroughly tested and are to be translated into native code, the debug option potentially causes much larger object code modules to be produced, because the code required to perform run time range checks almost always occurs right in the middle of a sequence of code which could have been highly optimized if the debug code were not present. The debug code itself is not particularly large, but the OS-9 Pascal native code translator products can, if allowed, compress many long sequences of native code into much smaller sequences. The OS-9 Pascal compiler produces pcode which was designed with the goal of native code optimization in mind, and the native code translators can do an impressive job for most programs. If you feel you need the debug code in the native code version of any procedure or program, then by all means use it — it is a very important and powerful feature. Just keep in mind that you can often pay a high price in memory requirements and execution speed. Debug code does not, by contrast, typically add significantly to either the memory requirements or execution time of the pcode version of programs. This is, again, due to the very way that the pcode was designed in the first place.

Designing Programs to Be Run By PascalS

If a program is going to be run using PascalS, the virtual code swapping interpreter, design the program with its particular performance considerations in mind. Code which is executed just once or relatively infrequently in a program should be placed in a separate procedure or procedures. That way it will seldom have to be “swapped in” for execution, thus allowing what swap buffers that are available to be better used for holding frequently used code segments. Remember that every time a byte of pcode is needed which is not currently in a swap buffer, some code which is currently in a swap buffer is going to be thrown away. A poorly designed program can cause a lot of swapping to occur which can significantly slow down the execution of a program.

Optimizing Variable Declarations

Declare any small and frequently used variables at the beginning of the VAR declaration part of any procedure or program. Both the pcode and the native code versions of a program can take advantage of variable allocations that are near to the stack mark within a stack frame. The two probably most important “thresholds” of address ranges are at -16 and -128. That is, data which is allocated to an address between 0 and -16 as shown on the compilation listing is very efficiently accessed. Data which is allocated to an address between -17 and -128 is slightly less efficiently accessed, and data allocated to locations below -128 are least efficiently accessed. These two thresholds just mentioned are by no means universally observed, and the list of addressing efficiency considerations would be long indeed if it were to be written out. You can, as a rule of thumb, remember that for the most memory and execution efficient programming you should declare the data which will be most frequently referenced during the execution of a procedure either at the end of a parameter list, if it is a passed parameter, or near the beginning of VAR declarations for other data types. Refer to the chapter on writing procedures in native code for more information on stack frames, stack marks, and variable allocation.

Accessing Absolute Memory Locations

If you need to refer to an absolute memory location within an OS-9 Pascal program, such as to access a hardware device's control registers, you need to do two things: disable the compile time debug option, and use overdefined records. The following code shows an example:

Figure 9.1. Code Example to Access Absolute Memory Locations.

```
PROGRAM memorydemo;
VAR
   trix: RECORD CASE boolean OF
            true : (i: ^integer);
            false: (p: integer);
         END;
   j: integer;
BEGIN
{$D-} { Disable run time range checking code generation }
trix.p:=$1234; { Setup "p" to access memory beginning }
               { at hex location 1234 }
j:=trix.i^;    { Get a 2 byte integer from locations }
               { 1234 and 1235 }
{$D+} { Re-enable run time range checking code generation }
END.
```

Debug code generation must be disabled for at least the small section of code as shown, since one of the things that the debug code does is to check that all memory references via pointer variables refer to memory which is within the program's data area. Since, presumably, you are trying to access some special location in memory, and it is unlikely that the special location could be in your program's data area, you must disable debug code generation for the above "trick" to work. Since the variable "`trix.p`" is an integer, it is perfectly legal to set it to the value "1234". Variable "`trix.i`", however, is a pointer to an integer, but it occupies the same 2 byte location as "`trix.p`". By executing the statement, "`j:=trix.i^`", you in fact say, "store in variable `j` the 2 byte integer which is pointed at by the variable `trix.i`". Refer to any good book on Pascal programming for further information on this type of program trick and on the use of overdefined variables in general.

Deleting Files

To cause any file to be deleted, simply issue a REWRITE call to the file. For example, if a program wants to open file "f" as shown in figure 2 below for update access, but it wants to be sure that it is a new file, then it could do the following:

Figure 9.2. Code Example to Delete a File.

```
PROGRAM deletedemo;
VAR
   f: FILE OF ARRAY [0..63] OF char;
BEGIN
rewrite(f);
update(f)
END.
```

You should keep in mind a couple of points. First, if calls to REWRITE and UPDATE do not supply a filename for a second argument as in the example above, then the file name used is the identifier name of the file in the source program (file name "f" for the example above), and the file is assumed to be in the current data directory. For the above example, then, the file named "f" in the current data directory is first deleted if it already exists. Next, a file named “f” is created in the current data directory and is opened for update access. The second point, as suggested by the above, is that with the call to REWRITE, you automatically delete any existing file of the same name, if it isn't write protected. If the file is write protected, then a run time error would occur when the attempted deletion was performed. Automatic deletion can be dangerous; data in the file is permanently lost.

Bit-Level Operations

The “arithmetic” operators “&”, “!”, and “#” perform the AND, OR, and EXCLUSIVE OR bit-by-bit logical operations, respectively, on integer variables. Also, the standard procedures FIELDPUT and FIELDGET can be used to implement a wide range of bit functions. Some examples are shown in figure 3 below:

Figure 9.3. Code Examples Using the Standard Procedures FIELDPUT and FIELDGET.

```
FIELDPUT(i, bit number, 1, 0);
    { Resets a bit in the integer variable "i" }
FIELDPUT(i, bit number, 1, 1);
    { Sets a bit in the integer variable "i" }

i:=ord(ch);                      { The following 3    }
FIELDPUT(i, bit number, 1, 0);   {statements reset a  }
ch:=chr(i);     { bit in the character variable "ch"  }

FIELDPUT(i, bit number, 1,
   1-FIELDGET(i, bit number, 1));
   { Flips a bit in the integer variable "i" }

i:=FIELDGET(i, 0, 16); { Rotate variable "i" 1 bit to }
                       { the right }
i:=FIELDGET(i, 14, 16};{ Rotate variable "i" 1 bit to }
                       { the left }
```

In addition to the obvious applications, FIELDPUT and FIELDGET can be used to tightly pack data in memory, set bits on and off in I/O device control registers, to extract 4-bit groups from a variable in order to print out the variable's value as a hexadecimal number, and for a host of other functions.

Using Zero-Base Array Indices

If it is not inconvenient, make array indices start at zero for the lower bound. When any reference is made to an element of an array, the index value is scaled so that the array looks as if it begins with an index of zero. For example, for the following two lines:

```
VAR a: array[2..5, -6..7] OF integer;
i:=a[3, -6];
```

the reference to row 3, column -6 of array “`a`” is scaled to reference row 1, column 0 of the actual array that is stored in memory. What this all means is that all arrays are stored in memory as if each index really begins with a lower bound of zero. Because of this mapping, any reference to an element of an array must be adjusted by run time code to conform to the real array mapping. If, however, an array index already has a lower bound of zero, then no scaling is needed - thus there are fewer instructions to be executed in the program, and the program is smaller. If there is some reason for not making an array index start at zero, then by all means don't. Remember that there is a very small speed and memory penalty for each reference if you don't.

Using the SHORTIO Standard Procedure

The SHORTIO standard procedure can be used to make copies of files of arbitrary lengths. Declare the input and output files of the copy program to be fixed length files of some identical length. Use the standard procedure IOABORT to disable error aborting for the input file. Read from the input file and write the resulting record to the output file until an I/O error occurs for the input file. If the IORESULT value for the input file is then 68, indicating that a GET was attempted on a short record, execute the statement:

```
SHORTIO(outputfile, linelength(inputfile));
```

and proceed to write a copy of the input record. The SHORTIO call sets the length of the next PUT call to the output file to the truncated length of the input file record. If the IORESULT value was 69, indicating a GET with EOF true, no SHORTIO call or PUT is needed. Any other IORESULT value indicates a programming or file system error.

Chapter 10. The Run-time Environment

SYNOPSIS: This section describes what happens inside the system while P-code or native code Pascal programs are running, including memory allocation, run-time options, and error handling.

What Happens When Programs Are Run

There are two ways to run Pascal programs: P-code programs are run using either the PascalN or PascalS P-code interpreters (See Chapter 3), or programs translated to native code by PascalT can be run directly from OS-9 Shell command lines (See Chapter 4).

As programs are executed, many operations are performed “behind the scenes” by either the P-code interpreter and/or the Pascal support package. This includes allocation of memory when procedures and functions are invoked, processing when errors occur, etc. Even though these functions are usually so automatic that novice programmers need not have an understanding of how they work, more sophisticated programs should be designed with consideration towards these factors, This is especially true of programs that use either Pascal-generated or hand-coded native code.

Pascal Memory Utilization

The requirements of the Pascal language require a fairly sophisticated scheme for management of data memory. An example is the recursive nature of functions and procedures that require allocation of separate variable storage areas for each procedure or function invocation. A good understanding of how Pascal utilizes memory can be valuable to advanced programmers in order to use all of the options and capabilities of OS-9 Pascal.

When a P-code interpreter or a compiled native code Pascal program is run, OS-9 allocates to the task a contiguous block of RAM memory for its working storage. This area is called a “data area”, and is for exclusive use of the Pascal task. There is a certain minimum size for the data area coded into the Pascal program's memory module header which can be explicitly expanded using the OS-9 Shell's “#” memory size option.

Pascal further internally subdivides the data area into several sections. Most of the space is used for variables declared in the Pascal program, but some space is also required for the Pascal runtime system internal use. If a P-code interpreter is being used, part of the data area must be used to hold the P-code or swap buffers for P-code; if the program is compiled native code, the machine language instructions are contained in a standard OS-9 “memory module” which is located outside the data area in a separate memory area. For this reason it is more efficient for two users running exactly the same Pascal program to do so using native code which can be shared. P-code programs cannot be shared so a separate copy must be loaded for each user.

The main subdivisions of the Pascal data area are:

1. HEAP MEMORY: Memory used for dynamic data structures using the NEW, MARK, and RELEASE standard procedures.
2. STANDARD I/O BUFFERS AND FCBs: data structured needed for the three predefined files INPUT, OUTPUT, and SYSERR.
3. GLOBAL VARIABLES: storage for the main program ("outer block") variables (local and extended stack).
4. LOCAL VARIABLE STACK: this stack is used to dynamically allocate variable storage (local and extended stack) for procedures as they are called during program execution.

5. P-CODE BUFFER: If a P-code interpreter is being used, this section holds the P-code for all procedures (PascalN), or swap buffers for P-code (PascalS). Native code programs don't have this section because the program is elsewhere.
6. DIRECT PAGE: This section is used for internal working variables of the Support package routines.

Example 10.1. Pascal Data Area Utilization Map

```
            +---------------------------+
            |                           |
high memory |        HEAP MEMORY        |  Used by NEW, MARK,
            | (Can be extended upward)  |  and RELEASE calls
            |                           |
            +---------------------------+
            |                           |
            |    STANDARD I/O BUFFERS   |  Buffers and FCBs for
            |   AND FILE CONTROL BLOCKS |  INPUT, OUTPUT, SYSERR
            |                           |
            +---------------------------+
            |                           |
            |     GLOBAL VARIABLES      |  Variable storage for
            |                           |  main procedure
            |                           |
            +---------------------------+
            |                           |
            |      LOCAL VARIABLE       |  Variable storage for
            |          STACK            |  all other procedures
            |                           |  (dynamically allocated)
            +---------------------------+
            |                           |
            |  PASCALN P-CODE AREA, or  |  Program storage; this
            |  PASCALS SWAP BUFFERS     |  area nonexistent for
            |                           |  native code programs
            +---------------------------+
            |                           |
            |        DIRECT PAGE        |  SUPPORT package working
            |                           |  storage
low address +---------------------------+
```

The diagram above shows the various sections of the Pascal data area. Usage of some sections is fairly self-explanatory, however, three sections are of particular interest to the programmer because they are used for program, variable storage. The are the HEAP, GLOBAL VARIABLE, and LOCAL VARIABLE sections and are discussed in some detail in the following pages.

Global and Local Variable Storage

All Pascal variables must be declared in a VAR statement, and storage for these variables is assigned in the GLOBAL VARIABLE and LOCAL VARIABLE STACK sections of the Pascal data area.

Variables declared in the main program (Procedure 0 or “outer block”) are global in scope with respect to all other procedures and are thus called “Global Variables”. Addresses of global variables are assigned to specific locations within the GLOBAL VARIABLE section. Because the main procedure cannot call itself, only one set of variable locations is required, i.e., dynamic allocation is not required.

Variables declared within all other procedures (Procedures 1..N) are local, meaning they are only known within the procedure in which they are declared and other procedures that it may call. To further complicate matters, Pascal permits “recursive” procedure calls where procedures can call themselves but with unique variables in each call. For these reasons, Pascal must use a stack to dynamically allocate storage to procedures when they are called, and deallocate storage when they exit to their caller. The stack comprises the LOCAL VARIABLE STACK section of the Pascal Data space.

Local and Extended Stacks

In addition to storage for variables for each procedure invocation, storage may also be needed for temporary working variables such as for holding intermediate results during evaluation of expressions. These temporary variables are automatically defined and managed by the compiler and are of no concern to the programmer except for the fact they do require memory space and must be taken into account when manually figuring and assigning memory space.

Variables declared in the program are kept on the “local stack”. Compiler-created temporary variables are kept on the “extended stack”. Both stacks are located in the section of the Pascal data area called the “LOCAL VARIABLE STACK”. The main procedure (Procedure 0) also may need an extended stack, but since it has its own global variable section part of it is reserved for the main procedure's extended stack.

While a program is running, variable storage is assigned from the stack when procedures are called and returned to the stack when they are exited. It is possible that the stack(s) may fill up so that another procedure invocation is not possible. The P-code interpreters check the stack size before each procedure invocation in order to make sure that sufficient stack space is available. Native code programs also include stack-checking code unless you specifically instruct otherwise.

If the stack becomes too full to perform another procedure call, a fatal error occurs and the program is aborted. You must then use the “`l`” and/or “`e`” run-time options to increase the size of the local stack and/or the extended stack as necessary. Because procedure calling depends on program flow which can be almost random at times, it may be impossible for you (or the compiler) to accurately predict worst-case requirements.

When a program is compiled, the procedure table printed at the end of the listing gives the exact number of bytes of storage required for each procedure's LOCAL variables and extended STACK (See Chapter 2). Therefore, the sum of the local stack and extended stack required by a procedure is the total amount of storage it requires (per call) within the local variable stack. The compiler sets the default size of the local variable stack section to a estimate based on these statistics. You can optionally use the “`l`” and “`e`” run-time options (See Chapter 11) to manually alter the size of the local variable stack area. For example, you might need to increase the size in the case of highly recursive programs where the compiler's estimate was too small, or if economical use of memory is crucial you may want to manually compute worst-case requirements in order to reduce the size of the compiler's too-generous estimate.

Each call to a procedure requires a block of memory called a “stack frame” to be allocated. A stack frame consists of the memory required to hold the local and extended stacks for a procedure plus an additional 7 bytes to hold a “stack mark”. Each local variable is assigned some number of bytes of memory depending on its structure. Characters and boolean type variables are assigned 1 byte each. Integers are assigned 2 bytes each. Real variables are assigned 5 bytes each. Arrays are usually assigned multiples of these quantities. For example, an array of 6 integers requires 6 times 2 bytes or 12 bytes of storage. Records are assigned storage according to their overall structure. A record that contains a real variable, an integer variable, and a boolean variable would require 5+241 bytes or 8 bytes of storage. Arrays of records require number-of-elements times record-size bytes of storage. Records which are allocated as local variables and which contain variants are allocated enough storage to contain the largest possible combination of variants.

If more than one variable is declared in a single declaration statement, the variables are allocated right to left. For the declaration:

```
i, j, k: integer;
```

the variable “`k`” is first assigned to the next lower 2 bytes of storage, then “`j`” is assigned to the next 2 lower bytes, then “`i`” to the next 2 lower bytes.

Heap Storage

Heap storage is for variables that are dynamically assigned, and it is managed via the standard procedures NEW, MARK, and RELEASE (These are described in Chapter 7). Only programs using these calls require HEAP section space; otherwise the size of this section may be zero. The run-time “`h`” option can be used to set the size (See Chapter 11).

MARK and RELEASE calls provide a way for saving and resetting the top of heap pointer. In particular, you can use MARK to save the current top of heap pointer, allocate several variables in heap via calls to NEW, and then use RELEASE to reset the top of heap pointer to what it was, so that the next call to NEW will reuse the heap space allocated since the last MARK call. This allows you to reuse heap memory over and over after a group of one or more variables is no longer needed.

Each call to NEW requests that some number of bytes of storage in the heap area be set aside for creating a variable, and the address of that assignment is returned to the pointer argument used in the NEW call. If the variable being allocated is a record with variants, the call to NEW must describe the specific values of all applicable variants to be used in computing the size of the record to actually allocate. In other words, a record type could be declared which requires 53 bytes of storage for some combination of variant values and requires 49 bytes for a different combination. If a variable was declared in local memory with this record type, the variable would be assigned 53 bytes of storage but by allocating the variable in the heap area copies of the variable can be created which use either 49 or 53 bytes.

If the amount of heap section space initially allocated is too small to handle a NEW call, Pascal will call OS-9 to request that the data area be given additional memory in order to expand the heap section space. If OS-9 can't do this for any reason, a fatal error occurs. When a RELEASE call gives the additional space back, Pascal automatically calls OS-9 to return the memory. The “`r`” run-time option (See Chapter 11) can be used to inhibit automatic return of addition memory if it is anticipated that it will be needed again later.

Chapter 11. Run-time Options

Programs produced by OS-9 Pascal accept an optional set of run time options on their command lines. These options control the way certain functions are handled while programs are being executed. Because the Pascal compiler is written in Pascal, these options also can be used to affect its execution.

When a compiled Pascal program is run, any combination of the options described in this section can be given on the command line used to run the program. These options are mostly handled by the “support” package, so the option apply both to P-code and native code programs.

If more than one option is specified, each option after the first must be separated from its predecessor by spaces and/or commas. If an option is specified twice, the last or rightmost specifications is used. If any errors are detected while scanning the option list, the program will be aborted. Upper and lower case letters are considered to be equivalent. For any option which allows a '+' or '-' modifier, the option will be accepted without the modifier and will be treated as if it had a '+' modifier.

Those options which have number (memory size) parameters can be given either a decimal number in the range of 1 to 65535, or a decimal number in the range of 1 to 63 followed by the letter “k” in which case the decimal number will be multiplied by 1024 to get the equivalent number of “KBytes”. An error message will be generated if any individual number is greater than 65535 or if the sum of the “l” and “e” values is greater than 65535.

These options determine various modes in effect at the time the program is started. The Pascal standard library includes procedures that can override the initial options and change modes under program control. In other words, options remain in effect until explicitly changed by the program. See Chapter 7 for detailed descriptions of these standard procedures.

Some example are:

Using Pcodefile and interpreters:

```
Pascaln Pcodef l10000 e500 j- i+
Pascals Pcodef L10000,E500 S20k i+
```

Using fully translated programs:

```
PgName j+ i+ a- #20k
PgName r+, h10, r+ #10k
```

j+ (default)
j-

“`j+`” sets the right justification mode, and “`j-`” sets the left justification mode for writing strings. This option exists because of a difference between the ISO specification which calls for right justification, and the original Wirth/Jensen specification which call for left justification. We have allowed OS-9 Pascal to work either way to aid in program portability.

For example, if a write statement were used to put the string variable “`STR1`”, an array of 20 characters, into a field with a width of 30 characters as in:

```
write(str1:30);
```

then 10 leading spaces would be written before the 20 character string. Strings which are written with width specifications that are wider than the string itself will be right justified within the field. This is how the ISO specification says how strings should be written. This options sets the default start up mode for string handling. The standard procedure RIGHTJUST can be used to change this option within a program.

m+ (default) m-	This option controls how the MOD function works and also exists due to differences between ISO and Wirth/Jensen specifications. We have allowed OS-9 Pascal to work either way to aid in program portability. “`m+`” directs Pascal to use the MOD algorithm as described in the ISO specification. The right argument must be a positive number and only a positive result or zero can be returned. “`m-`” directs Pascal to use the classic MOD algorithm which returns the signed remainder after performing signed division. The standard procedure ISOMOD can be used to change this option within a program.
a+ (default) a-	This option sets the default mode for arithmetic error handling. “`a+`” directs Pascal to abort the program and issue an error message if any arithmetic errors are detected. Arithmetic errors include overflow, divide by zero, and out of range arguments for mathematical standard functions. “`a-`” directs Pascal to not abort the program or issue any error message if any arithmetic errors are detected. For integer overflow errors, the least significant 16 bits of the result are retained. For real, overflow and underflow indeterminant results are retained. Divide by zero errors will return a zero result. Invalid function arguments will cause indeterminant results. This option is useful for programs which need to implement their own method of arithmetic checking or for performing algorithms which depend on residue results such as power congruence methods. The standard procedure MATHABORT can be used to change this option within a program.
r+ r- (default)	This option directs Pascal to retain heap memory (`r+`) obtained by the NEW procedure after the RELEASE standard procedure frees it, or to call OS-9 to return heap memory (`r-`) after RELEASE calls. Use of the “`r+`” option, when it is anticipated that more heap memory will be required later, will improve program efficiency since there will be fewer calls made to OS-9 to request memory. In a multi-tasking environment this option can also be useful because it can guarantee memory will be available later. See also “`h`” option below.
i+ i- (default)	The “`i+`” option tells Pascal to generate a memory statistics report. “`i-`” inhibits generation of the memory statistics report.
l <number>	This option overrides the default startup memory size for the local variable stack, which is used for all procedure and function variable storage. The stack is otherwise given a size based on the compiler's automatic estimate. This option is generally used with highly recursive programs that may need more variable size than usual. This option applies only to the PascalS and PascalN interpreters; stack size for pure native code procedure produced by PascalT are controlled by the OS-9 Shell “#” memory size option.
e <number>	This option overrides the default startup memory size for the extended stack, which is used for internal variables and variables used by native-code procedures declared in EXTERNAL statements. This option applies only to the PascalS and PascalN interpreters; extended stack size for pure native code procedure produced by PascalT are controlled by the OS-9 Shell “#” memory size option.

h <number>	Specifies the minimum amount of heap memory to be initially reserved for a program. Heap memory is used for dynamic data structures via the NEW, MARK, and RELEASE standard procedures (see Chapter 7). This minimum memory will be retained throughout the execution of the program. If more heap space is required, Pascal will request it from OS-9. See also "`r`" option.
s <number>	Specifies the size of the virtual code swapping area to be allocated. This option only applies to PascalS, the swapping P-code interpreter. The swapping area should be as large as possible to increase program execution speed (See Chapter 3 regarding PascalS and its virtual swapping system). The effect of the '`s`' option is described in the chapter on running user programs under the interpreters.

Finally you should note that some of the run time options are invalid or have no effect for some of the OS-9 Pascal products. The Pascal compiler itself, for instance, uses a highly specialized version of the virtual code swapping interpreter, and any attempt to specify the '`j`', '`m`', or '`a`' options will be flagged as an error. Also, any program which runs completely in native code, that is, it is not a pcode or a hybrid of pcode/native code, ignores any values supplied for the '`l`' or '`e`' options. Furthermore, purely native code programs cannot derive any significant number for the number of bytes of free heap memory on the memory statistics report. For pure native code programs, the effect of the '`l`' and '`e`' options is provided by the '`#`' option on the SHELL command line. See the OS-9 user and system manuals for more discussion of SHELL and its options.

Chapter 12. Run-time Error Handling

If during program execution an error occurs which causes your program to abort, the following type of error message is written to the system error path:

```
PASCAL ERROR #w
(error message text)
PROCEDURE #x0
PROCEDURE #x1
   .
   .
PROCEDURE #xn
LINE NUMBER=y
PCODE LOCATION=z
```

where:

“w” is the error number that occurred. Error numbers less than 100 generally refer to I/O errors where other numbers generally refer to process errors.

“error message text” Is the text of the message taken from the PASCALERRS file. If the PASCALERRS file cannot be opened, or if the appropriate text cannot be found, this line will be omitted. If the error is an I/O error, a second line will usually be displayed giving an OS-9 error message relating to the I/O error.

x0, x1, ..., xn is an unlinking of the procedure call nesting. It indicates that procedure number “x0” was executing when the error occurred, and the “x0” was called by “x1” and so on until “xn” is found which is the first procedure number which began execution. The “x” numbers refer to procedure numbers as shown in the procedure table list (see Chapter 2), Procedure number “xn” should always be zero if the stack hasn't been destroyed. The same procedure number can appear several times in the list if recursion has occurred. If a native code procedure is part of the call nesting, it will show up as procedure number 255 regardless of which native code procedure it might be; native code procedures lose their numeric identity as part of the requirement to be a native code procedure. If execution hasn't proceeded far enough for a call stack, to be built, or if the call stack is invalid, either no unlinking will be shown, or the unlinking may show meaningless procedure numbers.

“y” is the line number of the source program where the error occurred. If the source line number begins with the keyword “END” or "ELSE", it is possible that the error really occurred in the code for the next previous significant source line, or that the error really occurred in the termination code for whatever type of compound statement that the “END” or “ELSE” terminates. This error line is only given if you have enabled the inclusion of source line numbers in the source program (see Chapter 2). As with the previous error lines, this line is reported only if execution has proceeded far enough for a valid line number to be known. Also, if line number inclusion is selectively enabled and disabled within the source program, this error line shows the last known source line number.

“z” is the P-code location within procedure “x0” which was being executed when the error occurred. Where the line number message can quickly get you to the vicinity of the problem statement, the P-code location can give you an idea as to where within a statement the problem occurred. As with the previous error lines, this line is reported only if execution has proceeded far enough for a valid P-code location to be known. Furthermore, there are several errors which can occur for which the P-code location information gets lost, in which case this line will not appear. Most notably, many I/O errors and the address multiply overflow error cause the P-code location information to be unknown at error reporting time. Programs which are in their initial stages of testing should probably cause line numbers to be included in the P-code file to aid in debugging and testing. As the program becomes more reliable and error free, you might inhibit line number inclusion to achieve slightly enhanced execution and memory efficiency - relying solely on the P-code location report for further debugging purposes.

Most abort errors are broken up into two classes: input/output (I/O) and mathematical. If an abort error is in one of these two classes, it's abortive power can be disabled via either the IOABORT or the MATHABORT standard procedures. The standard functions IORESULT and MATHRESULT are provided so that you can deal with such errors within the program. Some programmers believe that you should never disable system checks for I/O or math errors. If you are one of these, then simply do not use any of the above named standard procedures. If, however, you are of the philosophy that production programs should be written with the idea that they should never be aborted by the system no matter what garbage the user feeds to the program, that the program should intelligently report the nature of the erroneous input, and, if necessary, gracefully shut down, then the above standard functions and procedures can be of great benefit. See the chapter on standard functions and procedures for more information on these routines.

There are, however, a few errors which always unconditionally abort your program. A case select error is one. This error occurs when a case statement is executed but there is no statement which has a constant selection list containing the required selection value. For example:

Figure 12.1. A Code Example Which Would Produce a Case Error.

```
i:=5;
CASE i OF
   0, 7: DoThisStatement;
   1..4: DoAnotherStatement
END; { CASE }
```

would cause a run time abort since the actual value of variable "`i`", which is 5 in this case, does not appear in any constant selection list. One of two actions can be taken to prevent this type of error — either make sure that all possible values of the selecting expression are accounted for in the constant selection lists, or use the OTHERWISE case statement option as in:

Figure 12.2. A Coding Example Which Eliminates the Case Error by Using the "OTHERWISE" Case Option.

```
i:= 5;
CASE i OF
   0, 7: DoThisStatement;
   1..4: DoAnotherStatement;
   OTHERWISE: BEGIN
              ReportError(i);
              GOTO EndOfProgram
              END
END; { CASE }
```

See the chapter on deviations and enhancements from ISO7185.1 for more discussion of the OTHERWISE option. Stack and heap overflow errors also unconditionally abort a program. Use of run time options can eliminate such errors, as is explained in the chapter on run time options. Address multiply overflow errors also unconditionally abort a program, but this, along with any other unconditionally aborting errors, can be avoided by good programming practices. The OS-9 Pascal system is intended to provide the ability to write programs which are as user friendly and user tolerant as possible and those errors remaining which always unconditionally abort a program do so because it is not feasible to intelligently recover from the error condition and continue with the program execution.

When using IOABORT and MATHABORT it is best to disable the abort process for the smallest range of code necessary. For example:

Figure 12.3. Code Example Using the Standard Procedure “IOABORT”.

```
ioabort(f, false);
reset(f, 'INPUTFILE');
ioabort(f, true);
i:=ioresult(f);
```

is a good way to check if a file already exists. If the file doesn't exist, variable “`i`” will contain the number 216 which is the OS-9 error number for an attempt to open a nonexistent file. If no errors were encountered while trying to open the file, variable “`i`” will contain the number 0. Remember that if multiple errors are encountered for a file between the IOABORT call which disables the abort process and the IOABORT call which re-enables the abort process, only the first error number is returned - the others are lost. The opposite holds true for math errors - only the last error number is retained, the others are lost. For example:

Figure 12.4. Code Example Using the Standard Procedure “MATHABORT”.

```
mathabort(false);
i:=32767+16000*16000+2;
mathabort(true);
j:=mathresult;
```

will set the variable “`j`” to number 199, indicating an integer overflow on add, subtract, or negate. The error caused by the integer overflow for the multiply is lost since the add operation which also caused an overflow occurred last. If, in the above example, the multiplication overflow caused the modulus result which is retained to be a negative number, it is likely that no overflow on add, subtract, or negate would have occurred. In that case, the variable “`j`” would be set to the number 184, indicating a multiplication overflow. Bear in mind the possible combination of events which can occur and the possible results for all such combinations when using IORESULT and MATHRESULT. Remember, also, that each time you call either IORESULT or MATHRESULT that the error number for the appropriate file or for the mathematical result is reset to zero. You must save a copy of the error number from the last call if you intend to use it more than once.

Chapter 13. Writing Assembly Language Procedures

SYNOPSIS: This section gives the general information you need to manually write external procedures (i.e. Pascal PROCEDURES and FUNCTIONS) in assembly language. This chapter can also help you understand the type of native code produced by PascalT, the native code translator. This section assumes you understand 6809 assembly language, assembly language programming under OS-9, and Pascal memory utilization as discussed in Chapter 10.

Variable Space: Stack Frames and Stack Marks

All variable storage used by a procedure invocation is allocated in a section of the Pascal stack called a “stack frame”. A stack frame consists of the following three components: a stack mark, local variables, and the extended stack. Any parameters which are to be passed to a procedure are part of the calling procedure's extended stack. Suppose that procedure Q is declared as:

```
FUNCTION Q(i: real; var j: real; ch: char): integer;
```

Then to call Q, the calling procedure must do the following in the order given:

1. Reserve 2 bytes on the stack to hold the integer result of Q.

2. Push the 5 byte value of variable '`i`' onto the stack.

3. Push the 2 byte address of variable '`j`' onto the stack.

4. Push the 1 byte value of variable '`ch`' onto the stack.

5. Load register B with Q's procedure number.

6. Build the first 2 bytes of the 7 byte stack mark by pushing onto the stack the address of the next higher (lexically) local data area.

7. Build the second 2 bytes of the 7 byte stack mark by executing a long branch to subroutine (LBSR) instruction to procedure Q.

Upon entry into procedure Q, the stack looks like figure 1 and the B register contains the called procedure's number.

Figure 13.1. Stack Contents After Calling Procedure Q.

```
            +-------------------------------------+
high memory ! integer result area (LSB)           !
            +-------------------------------------+
            ! integer result area (MSB)           !
            +-------------------------------------+
            ! value of variable 'i' (LSB)         !
            +-------------------------------------+
            ! value of variable 'i' (2nd LSB)     !
            +-------------------------------------+
            ! value of variable 'i' (3rd LSB)     !
            +-------------------------------------+
            ! value of variable 'i' (4th LSB)     !
            +-------------------------------------+
            ! value of variable 'i' (MSB)         !
            +-------------------------------------+
            ! value of variable 'j' (LSB)         !
            +-------------------------------------+
            ! value of variable 'j' (MSB)         !
            +-------------------------------------+
            ! value of variable 'ch'              !
            +-------------------------------------+
            ! address of next higher local data   !
            !     area (LSB)                      !
            +-------------------------------------+
            ! address of next higher local data   !
            !     area (MSB)                      !
            +-------------------------------------+
            ! return address (LSB)                !
            +-------------------------------------+
 low memory ! return address (MSB)                !
            +-------------------------------------+
```

Procedure Q then completes the 7 byte stack mark and does other setup with the code shown in figure 2.

Figure 13.2. Procedure Entry Code.

```
PROCQ  LDA   PROCN          LOAD WITH CALLING PROCEDURE
*                           NUMBER
       STB   PROCN          SAVE CALLED PROCEDURE NO.
       PSHS  A              PUSH CALLING PROCEDURE NO.
       PSHS  U              SAVE CURRENT COPY OP U-REG
       LEAU  ,S             NEW VALUE OP U-REG
       LEAX  -locsize,S     X REG POINTS TO REQUIRED
*                           BOTTOM OF STACK
       PSHS  X              PUSH PARAM FOR SUBR CALL
       LDD   #extsize       LOAD D-REG WITH EXTENDED
*                           STACK SIZE
       LDX   SUPJTBL        GET SUPPORT PACKAGE ADDR
       JSR   STACKCHK,X     ..DO STACK CHECK AND UPDATE
*                           THE STACK POINTER
       LDD   DHOLD          GET CURRENT VALUE OF DHOLD
       PSHS  D              SAVE COPY ON STACK
```

The code shown in figure 2 is virtually identical for every procedure entry. The only change from procedure to procedure is the label for the first line, the value of '`locsize`', and the value of '`extsize`'. There are 3 variables stored in page zero (i.e. in the base page) which must be used or maintained by every procedure. The variable `PROCN` is the one byte current procedure number. This variable must at all times indicate the procedure number as shown in the procedure table of the currently executing procedure. The variable `SUPJTBL` contains an absolute memory address of a known point within the support package. If the X register of the 6809 is loaded with this value, a '`JSR subr,X`' instruction, where 'subr' is the name of a support package subroutine as defined in the `PASCALDEFS` file, will cause the proper subroutine call to be made for any support subroutine. Variable `DHOLD` contains a 16 bit value which must be saved locally on every procedure entry and restored on every procedure exit. The value of '`locsize`' is, of course, the size as shown in the procedure table list for the procedure's local stack size.

After executing the code in figure 2, the 7 byte stack mark is completed. The form and contents of a stack mark is shown in figure 3. Figure 4 shows the form and contents of the stack and where the U and S registers point after a call is made to a procedure and the procedure entry code has been executed.

Figure 13.3. Format and Contents of a Stack Mark.

```
            +---+
high memory ! 6*! address of next higher      *indicates the
            !   ! local data area (LSB)        relative offset
            +---+                              within the 7 byte
            ! 5 ! address of next higher       stack mark.
            !   ! local data area (MSB)
            +---+
            ! 4 ! return address (LSB)
            +---+
            ! 3 ! return address (MSB)
            +---+
            ! 2 ! calling procedure number
            +---+
            ! 1 ! saved copy of U register (LSB)
            +---+
 low memory ! 0 ! saved copy of U register (MSB)
            +---+
```

Figure 13.4. Stack Contents During Procedure Execution.

```
            +---------------------------------------------------+
high memory ! result area (only if a function is being called) !
            +---------------------------------------------------+
            ! passed parameters (if any)     !
            +-------------------------------+
            ! stack mark                     !  --+
U-reg =>    +-------------------------------+    !
            ! local variables (if any)       !    -- stack frame
            +-------------------------------+    !
            ! copy of DHOLD                  !    !
S-reg =>    +-------------------------------+    !
 low memory ! available working stack        !  --+
            +-------------------------------+
```

References to most data is made via the U register. Passed parameters and the function result area are at a positive displacement from U, while local variables are at a negative displacement from U. The last parameter passed is stored beginning at '7,U' displacement for its most significant byte. The first local variable is stored at '-1,U' displacement for its least significant byte. To access data in lexically higher levels of procedures, that is in calling procedures, you need to do a 'basex' operation. A basex is done by climbing up the stack marks to find the address of the appropriate procedure's local data area. The address of the local data of the immediately calling procedure can be found in the 5th and 6th bytes of the current stack mark, as shown in figure 3. If you wanted to access a 2 byte variable which is at a displacement of -17 in the calling procedure's local data area, the following code would be used:

```
LDX   5,0      BASEX UP 1 LEVEL
LDD   -17,X    GET 2B FROM CALLING PROC'S AREA
```

Similarly, to access data from a procedure which is lexically two levels higher (i.e. the procedure which called the procedure which called the current procedure), two basex operations must be performed.

```
LDX   5,U      BASEX UP 1 LEVEL
LDX   5,X      BASEX UP 1 MORE LEVEL
LDD  -displacement,X  GET 2B OF DATA
```

To access data in the global area, that is data declared in the outer block, does not require basex operations to get to the proper level - the Y register always points to the top of global area. At a zero or positive displacement from Y is the system area which contains several pointers, buffers, work areas, and heap memory. At a negative displacement from Y is the outer block's local data - the global data area. Figure 5 shows the register contents and usage for procedures executing in an OS-9 Pascal environment.

Figure 13.5. Register Usage in an OS-9 Pascal Environment.

Register	Use
A	Available.
B	Available.
X	Available, also used for basex and general indexing operations.
Y	Points to the top of global, also the 1st byte of the system area.
U	Points to the top of local, also the 1st byte of the current stack mark.
S	Points to the current top of stack. Stacks are built going downward in memory.
DP	Points to the page zero data area. This area should never by used by a user program or procedure for its own data - the OS-9 Pascal support package manages this area.

The Y, U, S, and DP registers should normally never be used other than as shown in figure 5. If they are altered at any time, they must be restored before any calls are made to subroutine or before the procedure exits or calls any other procedure.

Calling Other Pascal Procedures

To call another Pascal procedure the following steps must be performed in the order shown:

1. Reserve any required stack space to hold the result if a function is being called.
2. Push any required parameter values and/or addresses onto the stack.
3. Build the first 2 bytes of the 7 byte stack mark by pushing onto the stack the address of the next higher (lexically) local data area.
4. If the procedure to be called is also known to exist in native code within the current module, load the B register with the number 255, otherwise, load the B register with the called procedure's number.
5. If the called procedure is not a native code procedure in the current module, jump to the interpreter's dispatch routine via:

```
LDX DISPATCH
JSR ,X
```

where DISPATCH is a page zero variable defined in the `PASCALDEFS` file. To call procedures which are also native code within the same module as the calling procedure simply use the branch to subroutine (e.g. LBSR) instruction to go to the procedure's entry code.

Finally, to exit from a procedure use the code shown below.

Figure 13.6. Procedure Exit Code.

```
        PULS  D           RESTORE THE SAVED COPY OF DHOLD
        STD   DHOLD
        LEAS  ,U          POINT S TO BOTTOM OF STACK MARK
        PULS  U           RESTORE CALLER'S U REG
        PULS  A,X         GET CALLING PROC NO. & RETURN ADR
        STA   PROCN       RESTORE CURRENT PROCEDURE NUMBER
        LEAS  2+paramsize,S  REMOVE LAST 2 BYTES OF THE
*                         STACKMARK AND ANY PASSED PARAMS
*                         FROM THE STACK
        JMP   ,X          ..RETURN TO CALLING PROCEDURE
```

Chapter 14. Conformance with ISO Standards

OS-9 Pascal Version 2.0 has some differences from the ISO7185.1 Level 0 language specification. Many of these are the result of enhancements made to this version of Pascal to improve its performance and to add programmer conveniences. If portability of programs written in OS-9 Pascal to other systems is important, these features should not be used. A complete list of deviations from the standard is given below:

1. CASE Statements can have an OTHERWISE option. (Non-standard enhancement).
2. Identifiers and numeric constants in the source program can contain underscore characters. (Non-standard enhancement).
3. The procedure directive EXTERNAL is permitted. (Non-standard enhancement).
4. Character strings can be up to 100 characters in length (Standard is not specific).
5. Source statements can be up to 110 characters in length. (Standard is not specific).
6. PROCEDUREs and FUNCTIONS cannot be passed by name; FILEs cannot be passed by value.
7. The attribute PACKED is ignored due to 6809 memory addressing. Variables are always allocated such that they are packed to the byte level. Packing at the bit level is not done. The standard procedures PACK and UNPACK are not implemented.
8. Set constants can be built using the subrange form “A..B” where “A” and “B” are constant members of the set being formed. (Non-standard enhancement).
9. A list of file names in the PROGRAM statement is not required, but if supplied, it is checked for correctness. (Non-standard enhancement).
10. DISPOSE (an ISO Level 1 feature) is not implemented.
11. Constants in the constant list of a case statement can have the form “A..B” which designates the list of values from A through B inclusive. (Non-standard enhancement).
12. Vectors, one dimensional arrays with character elements, can be indexed using the “expression FOR constant” form. (Non-standard enhancement).
13. GOTO statements may not reference a label outside the current procedure.
14. Standard procedures and run time options are provided to select either ISO Standard or Wirth/Jensen methods for string justification and the MOD algorithm.
15. Input and output operations are performed slightly differently for interactive or mass storage files. (Non-standard enhancement).
16. Each variable declared to be a file type has associated with it a file control block which is automatically initialized when the code block begins execution. The file itself is not automatically opened unless it is one of the three standard files: INPUT, OUTPUT, or SYSERR. When a block of code is exited, all files defined within that block are automatically closed.
17. A large number of standard functions and procedures have been added to the standard library. (Non-standard enhancement).
18. REALs use a five byte (9 1/2 decimal digit) format. (Standard defines this as implementation dependent; some Pascals have less precision).

19. Bit-by-bit boolean operators are included in OS-9 Pascal.

20. GOTO's are allowed in CASE, REPEAT, WHILE, IF, and FOR statements. (Non-standard enhancement).

21. Undefined or uninitialized variables and fields may be referenced without error.

22. A variable created by the variant form of NEW may be used as an operand in an expression.

23. PROCEDURE or FUNCTION declarations may be nested to 15 levels.

24. A variant field is allowed to be an actual variable parameter.

25. A file buffer variable may be passed as an actual variable parameter to a procedure which changes the current file position.

26. The EOLN standard function may be called for a file which has EOF set to true without error.

27. Record structures and arrays may not contain files.

28. The compiler does not issue warnings for out of bounds subrange conditional tests. This may generate code areas that will never be executed.

Appendix A. Error Message Descriptions

Following is the text of OS-9 Pascal error messages. Most messages consist of a number, followed by a colon, followed by a description. Those messages which have a number followed by an asterisk instead of a colon should not occur in Version 02.00.00 of the compiler but should not be changed either. Following the text of many of the error messages is a brief description of what the compiler was scanning or looking for which caused the error to occur, this will in many cases aid the user in correcting his program. in a few instances, further information is also given which indicates the most likely error and/or correction associated with the message.

1: Simple type expected.

> *When looking for a simple type, a valid starting token could not be found, i.e. a left parenthesis, plus sign, minus sign, integer constant, real constant, string constant, or an identifier.
>
> *When looking for a simple type, an identifier was found which has no valid type information yet determined.

2: Identifier expected.

> When scanning an enumeration declaration, the identifier found was not an identifier as part of the required identifier list.
>
> When scanning a field list within a record, a comma was found, but the token after was not an identifier.
>
> *With a record declaration, CASE was found, but the token after is not an identifier.
>
> When scanning a type declaration, up arrow was found, but the token after is not an identifier.
>
> When scanning CONST declarations, expected to find an identifier being declared.
>
> When scanning TYPE declarations, expected to find an identifier being declared.
>
> When scanning VAR declarations, expected to find an identifier begin declared.
>
> When scanning a procedure or function parameter list, found PROCEDURE or FUNCTION, but the token after is not an identifier.
>
> When scanning a function declaration, “:” was found, but it was not followed identifier naming the type of result.
>
> When scanning a procedure or function parameter list, found a list of identifiers followed by a colon, but the token after is not an identifier naming type of identifier(s).
>
> *PROCEDURE or FUNCTION is not followed by an identifier giving routine name.
>
> *A period is found indicating reference to a field within a record, but the period is not followed by an identifier naming the field.
>
> When scanning a parameter list, expected to find an identifier naming a parameter.
>
> *A procedure or function is being passed by name, compiler restriction does not allow this.
>
> *FOR is not followed by an identifier naming the control variable.
>
> *WITH is not followed by an identifier naming the record.

*PROGRAM is not followed by an identifier naming the program.

When scanning the list of file names after PROGRAM, an identifier naming a file was expected.

**Possible common programming errors which might trigger this message include:
- An extra comma was found in a list.
- A reserved word is used as a variable name.

3: PROGRAM expected.

*PROGRAM must be the first token in a program outside of any preceding comments.

4: Right parenthesis expected.

*Expected a right parenthesis here to balance a previous left parenthesis.

5: Colon expected.

When scanning a field list within a record declaration, expected a colon to terminate list.

When scanning a case selection either in a statement or in a record declaration, expected a colon to terminate selection list.

When scanning VAR declarations, expected a colon to terminate a list of identifiers being declared.

*Label number preceding a statement is not terminated by a colon.

When scanning a function declaration, expected a colon terminator to introduce the function type result.

6: Unexpected symbol found.

*Token scanned is not a valid next token for the type of scanning being performed. This is a sort of catch all error for whenever the scanned token doesn't fall within the set of tokens that the scanner knows how to handle next. Frequently, this error is preceded by other the error numbers, in this case, the error usually means that due to the other errors for this line the scanner has lost track of what should be happening next in the sentence. If there are no preceding errors for this sentence, then the token is simply invalid for syntax of the statement at this point.

**Possible common programming errors which might trigger this message include:
- A semicolon precedes ELSE in an IF
- This or previous statement is missing a required semicolon;
- An extra END is encountered.
- DO is used instead of BEGIN.
- Spaces are used within an identifier name.
- A keyword has been misspelled.
- A comment is malformed.
- A string or character constant is malformed.
- A malformed double character operator is found such as => instead of >= or = instead of := or one or more spaces or end of lines occurs between the first and second character.
- A comma is missing between elements in a list.
- An equal sign is used instead of a colon for declaring a record within a record.

7: Parameter list expected.

When scanning a procedure or function declaration, an unexpected token was found which is not a left parenthesis which would introduce a parameter list, or the token is not one which would validly terminate the declaration.

When scanning a statement, found a call to a procedure or function which requires a parameter list, but no list is found. When scanning a parameter list, found a semicolon which separates parameter items, but the next token is not a valid parameter list item.

8: OF expected.

*Within a record declaration, CASE followed by a tagfield is not followed by OF.

*While scanning a type declaration, ARRAY followed by bounds is not followed by OF.

*While scanning a type declaration, SET is not followed by OF.

*While scanning a type declaration, FILE is not followed by OF.

*While scanning a statement, CASE followed by a boolean expression is not followed by OF.

9: Left parenthesis expected.

When scanning a record declaration with a CASE part, a selection is found followed by a colon but is not followed by a left parenthesis which introduces the fields for the selection(s).

When scanning a statement, a procedure or function call is found which requires a parameter list, but no left parenthesis was found which introduces the parameter list.

10: Type expected.

*A type declaration was expected and an identifier was found for which no type information is known.

*A type declaration was expected and did not find a valid first token, i.e. an up arrow, PACKED, ARRAY, RECORD, SET, FILE, a left parenthesis, a plus sign, a minus sign, an integer constant, a real constant, a string constant, or an identifier.

When scanning a type declaration, found PACKED not followed by ARRAY, RECORD, SET, or FILE.

When scanning a function declaration, found a colon introducing the result type, but no type or an unknown type is found following the colon.

**Possible common programming errors which might trigger this message include:
- Using a reserved word as a variable name.
- Using a colon instead of equal sign in a type declaration.

11: Left bracket expected.

When scanning a type declaration/ found ARRAY but not a left bracket introducing the array bounds.

12: Right bracket expected.

*Expected a right bracket to match a previously found left bracket.

13: END expected.

When scanning a type declaration, found RECORD but not terminating END.

When scanning a case statement, cannot find terminating END.

*BEGIN is missing its terminating END.

14: Semicolon expected.

*Looking for a semicolon terminator and didn't find one here.

15: Integer expected.

When scanning a label declaration, expected an integer label number.

When scanning a WRITE or WRITELN call, found a colon introducing width or number of decimal places constraint which is not followed by an integer.

*Argument of ORD call or CHR call is not appropriate integer.

*Second and third parameter of FIELDGET call is not an integer expression.

*Second, third, and fourth parameter of FIELDPUT call is not an integer expression.

*Argument of SYSERR is not an integer expression.

*GOTO is not followed by an integer label number.

16: Equal sign expected.

When scanning a CONST declaration, identifier is not followed by an equal sign.

When scanning a type declaration, identifier is not followed by an equal sign.

17: BEGIN expected.

*A declaration part was scanned which is not part of an empty outer block, but the declaration part is not followed by BEGIN.

18: Invalid declaration part.

*A declaration part was scanned, but it is not followed by BEGIN or a program terminating period.

19: Field list expected.

When scanning a record declaration, RECORD is not followed by an identifier introducing a field list or CASE.

20: Comma expected.

*For this procedure or function call another parameter is expected, and the separating comma is not found.

When scanning the list of file names following PROGRAM, the last file name is not followed by a comma or right parenthesis.

21: Program terminating period expected here.

*Last token in a Pascal program must be a period after the final END.

22: Invalid ASCII character.

*A character was found in the program text outside of a string constant which is numerically less than a space character or numerically greater than a tilde character.

23: Expected a hex digit.

When scanning a hexadecimal number, found the dollar sign which indicates the start of a hexadecimal number, but the current character being scanned should be and is not a hexadecimal digit (0 through 9, 'A' through 'F', or 'a' through 'f').

24: Double period expected.

*Subrange simple type is being scanned, a double period is expected to separate low range from high range.

25: *COMPILER ERROR* Standard procedure number unknown.

> *A code generator routine in the compiler was passed an invalid number. Check system for memory problems first, then check for disk problems, then try using a fresh version of the compiler from a backup copy of the disk. If problem still persists, consult your nearest Radio Shack store.

26: Comma or colon expected.

> When scanning a record declaration, found a fieldlist, but the next token in not a comma to separate identifiers or a colon to terminate identifier list.
>
> When scanning a parameter list, found a list of identifiers, but the next token is not a comma to separate identifiers or a colon to terminate identifier list.

27: Constant is out of range.

> *Compile time check of allowable range for constant shows that the constant is out of range.

28: Identifier, VAR, PROCEDURE, or FUNCTION expected.

> When scanning a procedure or function parameter list, didn't find a valid next token.

29: PROCEDURE or FUNCTION not allowed here (compiler restriction).

> When scanning a procedure or function parameter list, found attempt to pass a procedure or function by name, this version of the compiler does not allow this.

30: Error in real constant, digit expected.

> *A string of digits was found followed by a period (but not a double period), but no digit was found after the period, this is an invalid real number according to the language spec. You may need to put a zero after the period.
>
> * A real number was found which ends with an “e” or “E” followed by an optional sign but not followed by a digit. After the “E” must come an integer exponent value.

31: String constant must be contained on a single source line.

> *An opening single quote character was found which indicates the start of a string constant, but the matching ending single quote character was not found on the same line of source. All strings must be fully contained on a single line.

32: Integer constant exceeds range.

> *An integer constant was scanned which has a value greater than 32767.

33: Too many nested scopes of identifiers (compiler restriction).

> When scanning a type declaration, found RECORD, but the symbol table cannot be further nested.
>
> When scanning a procedure or function declaration, the symbol table cannot be further nested.
>
> When scanning a WITH statement, the symbol table cannot be further nested.

34: Too many nested procedures and/or functions (compiler restriction).

> *PROCEDURE or FUNCTION found, but the symbol table cannot be further nested.

35: Invalid scope of name due to previous use.

> *The name in question has been previously used within the same block. An example of this is if a name was used as a constant in the outer program and the constant referenced within a subprocedure. Then a new type was declared using the same name as a member of a subrange type.

36: Too many errors detected for this source line.

*Ten or more errors were detected during the scanning of this source line. All errors past the ninth are ignored. Usually when this many errors are detected for a single line of source, it is because some other error has triggered secondary errors. If this is the case, correcting the primary error will get rid of the secondary errors.

37: Division by zero attempted.

*Either a compile time or run time process determined that a division by zero is being attempted.

38: Constant value must be greater than zero.

*An array reference using the <expression> FOR <count> language extension is scanned but <count> is zero or negative - it must be greater than or equal to one.

39: Element expression is out of range.

*An element of a set was scanned which is found to have a value less than zero or greater than 255.

40; SHORTIO called with record size out of range.

*Program called the standard procedure SHORTIO, but the value of the record size parameter is less than zero, or greater than the declared record size of the file. If the I/O abort flag is enabled for the file, the program will abort.

41: FILESIZE called with file not open.

*Program called the standard procedure FILESIZE, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

42: OS-9 error on FILESIZE call, OS-9 error number follows.

*Program called the standard procedure FILESIZE, and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

43: Undefined FORWARD procedures or functions found.

*A procedure or function was declared with the attribute FORWARD, but the body of the procedure or function was not found by the time is was required to be found.

44: Unimplemented file function in support package.

*You are running a program using either the “Support1” or “Support2” support modules which contain subsets of the full support module, and one of the missing support routines is needed. Cause the correct support module to be used as described in the implementation guide, and rerun the program.

45: IOABORT called with TRUE argument value, previous error not cleared.

*Once IOABORT has been called with a FALSE value, the I/O errors must be cleared by a call to IORESULT. The result buffer must be cleared before attempting to call IOABORT with a TRUE value.

46: Case selector is out of range of base type.

*Within a TYPE declaration a variant record entry has been declared with a value number which is not in range of the record case determinator type. An example is to declare a record with a variant record whose selector is a subrange. One of the possible record formats then uses a label which is out of the selector variable's subrange.

47: Label not in range of 0 thru 9999.

A label has been declared which is not in the range of 0 thru 9999.

48: Can't use NIL in a CONST declaration.

NIL is a reserved word which is used in reference to pointers bat does not have a value which may be assigned within a CONST declaration.

49: Defined function is not assigned a value.

When a function is declared, the name of the function must be the target of an assignment within the body of the function. All functions must return a value.

50: Constant expected.

*Looking for a constant, but did not find a valid token to start a constant, i.e. a plus sign / minus sign, integer constant, real constant, string constant, or an identifier.

**Possible common programming errors which might trigger this message include:
- Three periods in a row are found where a double period is intended.
- Finding a list when an array index specification is needed.

51: ':=' expected.

*A statement is being scanned, an identifier was found which is not a procedure or function name, therefore, it must be an assignment statement, but the next token is not ':='.

*A FOR statement is begin scanned, FOR was followed by an identifier name but was not further followed by ':='.

**Possible common programming errors which might trigger this message include:
- An equal sign is found where := is intended.
- An identifier is misspelled.

52: THEN expected.

*An IF statement is being scanned. IF was followed by a boolean expression but the next token is not THEN.

53: UNTIL expected.

*A repeat statement is being scanned, REPEAT followed by a statement list was found, but the terminating UNTIL was not found.

54: DO expected.

*A WHILE statement is being scanned. WHILE was followed by a boolean expression but the next token is not DO.

*A FOR statement is being scanned. FOR <ident>:=<expr>

TO/DOWNTO <expr> was found, but the next token is not DO.

*A WITH statement is being scanned. WITH was followed by an identifier, but the next token is not DO.

55: TO or DOWNTO expected.

*A FOR statement is begin scanned. FOR <ident>:=<expr> was found, but the next token is not TO or DOWNTO.

56: Duplicate identifier in PROGRAM statement parameters.

Within the PROGRAM statement parameters list an identifier has been used more than once.

57: Field width parameter negative.

When printing variables using WRITE or WRITELN a field width parameter may be used to specify the number of columns in which to print the variable. The width parameter may not be negative.

58: Factor expected.

*Looking for a factor but didn't find a valid first token, i.e. an integer constant, real constant, string constant, identifier, left parenthesis, left bracket, or NOT.

**Possible common programming errors which might trigger this message include:
- A real constant is intended but there are no digits before the decimal point.
- A malformed string constant is found.
- A malformed double character operator is found (see error message 6 for examples).

59: Variable expected.

When scanning a sentence, a variable reference followed by up arrow, period, or left bracket was expected.

60: Attempted to load or store the value of an address.

*While attempting to generate a pcode instruction, a form of object reference was found which would require the loading or storing of an address as a value.

61: Expected a file name.

*A procedure or function call is being scanned which requires as a parameter the name of a file here.

62: Invalid ordering of declaration parts.

Declarations are required to be in the standard order of LABEL, CONST, TYPE, VAR, and PROCEDURES or FUNCTIONS.

63: OS-9 error on file close, OS-9 error number follows.

*Program has either implicitly or explicitly closed a file and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

64: Read called with EOF true.

*Program called the standard procedure READ, but the file pointer is at EOF. If the I/O abort flag is enabled for the file, the program will abort.

65: Read called with file not in inspection mode.

*Program called the standard procedure READ, but the file is opened for output only. If the I/O abort flag is enabled for the file, the program will abort.

66: Read called with file not open.

*Program called the standard procedure READ, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

67: OS-9 error on file read. OS-9 error number follows.

*Program is attempting to read a file, but OS-9 detected an error. The I/O abort flag is enabled, so the program aborts and the OS-9 error is displayed.

68: GET attempted on short record.

*A GET is issued against a file of fixed length records, but the record found does not contain enough characters. If the I/O abort flag is enabled for the file, the program will abort.

69: GET called with EOF true.

*Program called the standard procedure GET, but the file pointer is at EOF, and the preceding operation to the file was not a call to REPOSITION. If the I/O abort flag is enabled for the file, the program will abort.

70: GET called with file not in inspection mode.

*Program called the standard procedure GET, but the file is opened for output only. If the I/O abort flag is enabled for the file, the program will abort.

71: GET called with file not open.

*Program called the standard procedure GET, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

72: Multiple decimal points encountered during READ of real number.

*Program called the standard procedure READ to get a real number, but two decimal points were found. If the I/O abort flag is enabled for the file, the program will abort. Check the format of the data being read for bad characters or missing terminator (i.e. a space) between numbers.

73: Digit expected during READ of real number.

*Program called the standard procedure READ to yet a real number, but at some point a digit was required and not found. Check the format of the data being read for bad characters or for a number with no digits before 'E' or no digits after 'E' with its optional sign.

74: Floating point overflow during READ of real number.

*Program called the standard procedure READ to get a real number, but the number is too large (bigger than approximately 1E37).

75: WRITEEOF called with record number out of range.

*Program called the standard procedure WRITEEOF, but the current record number is too big (bigger than approximately 2**31).

76: OS-9 error on WRITEEOF call, OS-9 error number follows.

*Program called standard procedure WRITEEOF, and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

77: WRITEEOF called with file in inspection mode.

*Program called the standard procedure WRITEEOF, but the file is opened for input only. if the I/O abort flag is enabled for the file, the program will abort.

78: Invalid character found for READ of integer.

*Program called standard procedure READ to read an integer and an unrecognizable character was found (i.e. not an optional leading plus or minus sign followed by a string of digits). If the I/O abort flag is enabled for the file, the program will abort.

79: Integer overflow for READ of integer.

*Program called standard procedure READ to read an integer, but the number is too large (i.e. not in the closed interval [-32768, +32767]. If the I/O abort flag is enabled for the file, the program will abort.

80: SEEKEOF called with invalid file size.

*Program called standard procedure SEEKEOF, but the file size in bytes is not an integral multiple of the record length for the file. That is, the file does not contain proper fixed length records as the last record in the file is too short. If the I/O abort flag is enabled for the file, the program will abort.

81: OS-9 error on SEEKEOF call, OS-9 error number follows.

*Program called standard procedure SEEKEOF and OS-9 detected an error. The I/O abort flag is enabled so the program aborts, and the OS-9 error is displayed.

82: SEEKEOF called with file not open.

*Program called standard procedure SEEKEOF, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

83: OS-9 error on file seek, OS-9 error number follows.

*program called standard procedure REPOSITION and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

84: REPOSITION called with record number out of range.

*Program called standard procedure REPOSITION, but the destination record number does not exist in the file. If the I/O abort flag is enabled for the file, the program will abort.

85: GETINFO or PUTINFO called with file not open.

*Program called standard procedure GETINFO or PUTINFO, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

86: REPOSITION called with file not open.

*Program called standard procedure REPOSITION, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

87: OS-9 error on GETINFO or PUTINFO call, OS-9 error number follows.

*Program called standard procedure GETINFO or PUTINFO and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 euro: is displayed.

88: POSITION called with file not open.

*Program called standard function POSITION, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

89: WRITEEOF called with file not open.

*Program called standard procedure WRITEEOF, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

90* *

91: INTERACTIVE called with file not open.

*Program called standard procedure INTERACTIVE, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

92: OS-9 error on file write, OS-9 error number follows.

*Program is attempting to write a file but OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

93: PUT called with EOF false.

*Program called standard procedure PUT but the file pointer is not currently at EOF and the preceding operation to the file was not a call to REPOSITION. If the I/O abort flag is enabled for the file, the program will abort.

94: PUT called with file not in generate mode.

*Program called the standard procedure PUT, but the file is opened for input only. If the I/O abort flag is enabled for the file, the program will abort.

95: PUT called with file not open.

*Program called the standard procedure PUT, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

96: OS-9 error on status call, OS-9 error number follows.

*Program has either explicitly or implicitly done a get or put status system call and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

97: SHORTIO called with file not open.

*Program called standard procedure SHORTIO, but the file has not been opened. If the I/O abort flag is enabled for the file, the program will abort.

98: OS-9 error on file open, OS-9 error number follows.

*Program has either explicitly or implicitly attempted to open a file, and OS-9 detected an error. The I/O abort flag is enabled, so the program aborts and the OS-9 error is displayed.

99: OS-9 error on file rewind, OS-9 error number follows.

*Program has implicitly attempted to rewind a file, i.e. it has issued a RESET, REWRITE, or UPDATE call to a file which is already opened and did not supply a new file name, and OS-9 detected an error. The I/O abort flag is enabled so the program aborts and the OS-9 error is displayed.

100: String or character array expected.

*The following standard procedures require a string or character array as their second arguments: RESET, REWRITE, UPDATE.

*The following standard procedures require a string or character array as their argument: SHELL, CNVTREAL.

101: Identifier is already declared.

*A new name is being declared, but the name already has been declared.

**Possible common programming errors which might trigger this message include:
- An identifier is misspelled.
- An identifier is being used both as a simple type and as an enumeration identifier.

102: Low bound exceeds high bound.

When scanning a subrange and the low bound of the range is found to be numerically higher than the high bound.

103: Identifier is not of the appropriate class.

*The identifier named is does not have valid attributes for use here.

**Possible common programming errors which might trigger this message include:
- A string or character constant is malformed.
- Previous errors were found in the declaration of the identifier.

104: Identifier is not declared.

*The identifier found has not been previously defined and this is not a forward pointer declaration.

**Possible common programming errors which might trigger this message include:
- An identifier is misspelled.
- A keyword is being used as an identifier name.
- A string or character constant is malformed.

105: Sign is not allowed here.

*When looking for a constant, a sign followed by an identifier name was found, but the identifier is not equivalent to a real or integer constant.

106: A number is expected.

*When looking for a constant, a non-string was found which does not resolve to a numeric value.

107: Incompatible subrange types.

When scanning a simple type, found a subrange, but the token to the right of the '..' is not compatible with the token to the left.

108: FILE is not allowed here.

When scanning a type declaration, found a pointer to a file. Pointers to files are invalid.

When scanning a type declaration, found FILE OF FILE. Files of file are invalid.

109: Type must not be real.

When scanning a subrange declaration, found a real range constraint. Only integer ranges are allowed.

When scanning a case discriminant (tagfield identifier), found a real-type identifier.

When scanning array declaration, found ARRAY [REAL].

110: Tagfield must be a scalar or subrange.

When scanning a case discriminant (tagfield) identifier, found an identifier which was not a scalar-type or subrange-type.

111: Incompatible with tagfield type.

When scanning a case selection list, the type of the selection value is Incompatible with the type of the case discriminant (tagfield).

112: Unimplemented Support Package function.

*The version of the Support Package which is in memory is not the version required for running this program. For example, your program may require floating point arithmetic, and the version of the Support Package which was either found in memory or which was loaded into memory does not support floating point arithmetic. There are three versions of the support package supplied. Either you should unlink the version of Support which is currently loaded in memory and load the required version before re-running this program or, if Support is not currently loaded into memory, you should cause the required version of the support package to have the file name “SUPPORT” in the current execution directory before re-running this program.

113: Index type must be a scalar or a subrange.

When scanning an array bounds declaration, the type of the bound given is not a scalar-type or a subrange-type.

114: Base type must not be real.

When scanning a set type declaration, found SET OF REAL.

115: Base type must be a scalar or a subrange.

When scanning a set type declaration, the type of the set being declared is not a scalar-type or a subrange-type.

116: Error in type of standard procedure parameter.

When scanning a call of a standard routine, the parameter found is not a valid type. See the user manual section on standard functions and procedures for a description of valid parameters.

117: Unsatisfied forward reference.

*The list of identifiers following were previously declared to be forward but their actual declarations were not found.

118: Machine code calls pcode but pcode doesn't exist.

*A native code routine erroneously calls a routine that is supposed to exist in pcode form but the pcode routine cannot be found.

**Usually this error is caused by partially translating a program, making changes to the program, and then doing a full translation on the program. During the first translation a routine was called by one of the translated routines which existed in pcode form. During the second translation the referenced routine was also converted to machine code but the first module still thinks it exists in pcode form.

119: Procedure was declared FORWARD, repetition of parameter list not allowed.

When scanning a procedure declaration and found a left parenthesis which introduces a parameter list but the routine was previously declared as forward and the previous declaration is the only place the the parameter list can be declared.

120: Function result type must be a scalar, a subrange, or a pointer.

When scanning a function declaration and a colon was found introducing a type for the function result but the token following the colon is not a scalar-type or a subrange-type, or a pointer-type.

121: FILE parameter cannot be passed by value.

When scanning a procedure or function parameter list and encountered attempt to pass a file by value, files can only be passed by name.

122: Function was declared forward, repetition of parameter list is not allowed.

When scanning a function declaration and encountered a left parenthesis which introduces a parameter list but the function was previously declared as forward and the previous declaration is the only place that the parameter list can appear.

123: Missing result type in FUNCTION declaration.

When scanning a function declaration and did not find a colon which introduces the result type of the function.

124: F-format allowed for real values only.

When scanning parameter list for WRITE or WRITELN call and found extraneous colon introducing a decimal width constraint but such a constraint is valid only for printing a real value and the item being printed is not real-type.

125: Not enough parameters given.

When scanning a call to READ or WRITE and no parameters are given, either give a file name and/or a list of items or use READLN or WRITELN.

126: Number of parameters does not agree with the declaration.

When scanning a procedure or function call and the number of parameters given does not agree with the declaration. For user routines look at the previous declaration. For standard routines see the section in the user manual describing standard procedures and functions.

127: Pcode file was produced by wrong series of compiler.

*The pcode file was produced by a compiler who's release level was not compatible with the current support package or interpreter. In general Pascal programs should be run with the support packages and/or interpreters which were the current versions when the program was compiled.

128: Result type of parameter function does not agree with its declaration.

When scanning a procedure or function call and the parameter being passed is a function which does not have the correct result type.

129: Operands are of incompatible types.

*Left argument of IN operator is not compatible with the right argument.

*Left argument of a relational operator is not compatible with the right argument.

*Left argument of the ':=' operator is not compatible with the right argument.

**Possible common programming errors which might trigger this message include:
- Attempting to assign a real result to an integer (use TRUNC or ROUND).
- Using a slash character instead of DIV.
- Using a string constant which is not the exact size required.
- An identifier is misspelled.

130: Expression must be a SET type.

*Right argument of IN operator must be a set-type.

131: Only equality and inequality test allowed for this type of operand.

*The following object types may be compared only as equal or not equal: pointer-type, record-type, and non-character array-type.

132: Strict inclusion test not allowed.

*Between sets only the following relationals are allowed: '<=', '>=', '=', '<>'.

133: Comparison of file types is not allowed.

*No relational operators can be applied to file names.

134: Illegal type of operand(s).

*Asterisk can only be used to multiply integers and/or reals or to intersect sets.

*Slash can only be used to divide integers and/or reals.

*DIV can only be used to divide integers.

*MOD can only be used to divide integers.

*AND can only be used to disjunct booleans.

*OR can only be used to conjunct booleans.

*Plus can only be used as a unary sign for reals or integers or to add reals and/or integers or to union sets.

*Minus can only be used as a unary sign for reals or: integers or to subtract reals and/or integers or to intersect sets.

**Possible common programming errors which might trigger this message include:
- Errors were encountered in the previous declaration of the identifier.
- Malformed double character operator (see error message 6 for examples).

135: Type of operand must be BOOLEAN.

*Expression after NOT must be boolean type.

136: Element type of a set must be a scalar or a subrange.

*After a left bracket introducing a set construction was found, a token is found which is not a scalar or subrange and thus cannot be a member of a set.

137: Element type is not compatible with the set.

*An element of a set was scanned but the element is not a member of the base set.

138: Attempt to index a non-array variable.

*A left bracket was scanned introducing an indexing expression but the item to the left of the left bracket is not an array-type.

139: Index type is not compatible with the declaration.

*Scanned an index expression but the type of the expression is not compatible with the array object being indexed.

**Possible common programming errors which might trigger this message include:
- Errors detected during previous declaration of identifier.
- Identifier is misspelled.

140: Attempt to select a field of a non-record variable.

*A period wan scanned introducing a field reference but the item to the left of the period is not a record-type.

141: Type of variable must be a FILE or a POINTER.

*An up arrow was scanned introducing a pointer reference but the item to the left of the up arrow is not a file-type or a pointer-type.

142: Illegal parameter substitution.

When scanning a procedure or function call and the parameter scanned does not have the same type as declared in the function/procedure declaration. For user routines, see the previous declaration. For standard routines, see the section in the user manual describing standard procedures and functions.

143: Illegal type of control loop variable, must be a scalar or subrange.

*Scanning a FOR statement and the identifier following FOR is not scalar-type or subrange-type.

144: Illegal type of expression.

*Scanning a case statement and the expression following CASE is not a non-real scalar-type.

145: Type conflict.

*Scanning a FOR statement, found FOR <ident>:=<expr> and possibly TO/DOWNTO <expr> but <expr> is not compatible with the type of <ident>.

**Possible common programming errors which might trigger this message include:
- Errors detected during previous declaration of identifier.
- Identifier is misspelled.

146: Assignment of files is not allowed.

*Item to the left of ':=' in an assignment statement cannot be file-type.

147: Case selection type is incompatible with selecting expression.

*Scanning a case statement, found a selection value but the type of the value is not compatible with the type of the expression following CASE.

148: Subrange bounds must be scalar.

*Scanning a subrange simple type and a string is found.

149: Index type must not be INTEGER.

*Scanning a type declaration and found ARRAY [INTEGER].

150: Assignment to a standard function is not allowed.

*Scanning an assignment statement and object to the left of ':=' is a name of a standard function.

151: Assignment to a formal function is not allowed.

*Attempt to assign a value to a formal <ident> which is a function name.

152: No such field in this record.

*Scanned a period which introduces an identifier which is a field name within a record but the identifier does not name a valid field name within the referenced record, check spelling of identifier or previous declaration of record.

153: Set range error.

It is possible to have different sets which are, for example, both integer subranges. These sets may have overlapping areas in which subsets of one may be assigned to the second set. The Set range error will be produced if a set is being assigned a value out of its subrange.

154: Actual parameter must be a variable name.

*Scanning a procedure or function call and the parameter being scanned must be passed by name but the token found is not a variable name.

155: Control variable must not be declared at an intermediate level.

*Scanning a FOR statement and the identifier of the control variable is not declared local to this block. Pascal language specification says that the control variable of a FOR statement must not be a formal parameter or be a variable declared at a different level than the block containing the FOR statement.

156: Multidefined case label.

*Scanning a case statement and a case selection or OTHERWISE appears more than once.

157: Too many cases in case statement (compiler restriction).

*Scanning a case statement and found more than 1000 selections.

158: Missing corresponding variant declaration.

*Scanning call to standard procedure NEW and token after a comma does not name a valid value of a variant. Check spelling of token, correct nesting of variant specifications for this call, or previous record declaration.

159: Real or string tagfields are not allowed.

*Scanning call to standard procedure NEW and token after a comma is real-type or string-type.

160: Previous declaration was not FORWARD.

*Scanning a procedure or function declaration but routine has been previously declared and did not have FORWARD attribute.

161: Attempt to declare FORWARD again.

*Scanning a procedure or function declaration and found FORWARD or EXTERNAL but routine has been previously declared with one of these attributes.

162: Parameter must be a valid variant value.

*Scanning call to standard procedure NEW and token after a comma is not a valid tagfield-type.

163: Null string not allowed.

Strings are defined to be a string of 2 or more characters enclosed within a pair of single quotes.

164: Lexical separation error, number terminated with a letter.

Found a number which was terminated by an alphabetic letter. Letters are not proper terminators for a number.

165: Label is already defined.

*Scanning a statement, found a label preceding the statement but this label has already been encountered in this block.

166: Label is already declared.

*Scanning a LABEL declaration and label number scanned has already been declared a label in this block.

167: Undeclared label.

*Scanning a GOTO statement and label number scanned is unknown in this block.

*Scanning a statement, found a label preceding the statement but this label has not been declared in a LABEL declaration for this block.

168: Undefined label.

*The label number(s) following were declared in a LABEL declaration for this block but were never encountered.

169: Error in base set.

*Scanning a type declaration, found SET OF INTEGER, integer sets not allowed - use SET OF CHAR or SET OF subrange-type.

170* *

171: Standard file was redeclared.

*Scanning a VAR declaration which is file-type and found attempt to redefine files INPUT, OUTPUT, or SYSERR.

172: Unknown or recursive type reference.

An unknown type has been referenced within a type or a type has used itself within its own type definition.

173: External procedure or function is expected.

*Scanning a procedure or function parameter list and found an unknown identifier which is assumed to be an external procedure or function name but no external declaration has been found. Check spelling of identifier or declare the external routine or declare the identifier in a TYPE declaration.

174: For control variable cannot be assigned a value.

The control variable for a FOR loop can not be the target of an assignment statement within the body of the for loop.

175* *

176* *

177: Assignment to function identifier is not allowed here.

*Scanning an assignment statement and object of assignment is the name of a function but the function declaration is not this block. Assignments to function names can only be made inside the block which declares the function.

178: Record variant is already defined.

*Scanning a record declaration and within a CASE a selection has been scanned which has already been selected.

179: Unimplemented language feature.

*Scanning a PACK or UNPACK statement. These routines are not implemented in this compiler.

180* *

181* *

182: OS-9 error on get heap call, OS-9 error number follows.

*Program called standard procedure NEW and OS-9 detected an error (usually not enough memory). Program aborts and displays OS-9 error. Check to see that memory is not fragmented when program begins execution or use the runtime “`h`” option to assure than enough heap memory will be available during program execution. This error typically indicates that either not enough heap area memory is available to start with due to too large a need of heap memory or due to multi-tasking activity or that heap memory which may have been available previously has since been returned to the system and is not available now. This latter cause can be eliminated by using the “`r`” or retain-heap run time option.

183: Integer divide by zero.

184: Integer multiply overflow.

185: Address multiply overflow.

*During computation of an address (typically do to array indexing or accessing a field within a record), an address greater than 65535 was generated. Probably the best method for pinpointing the cause of this error is to use the compile time debug option (`D+`) or more precisely don't disable it anywhere in your source code since it is enabled by default. With the debug option it is likely that the addressing error will show up as a range or indexing error and the runtime diagnostic will pinpoint the pcode location causing the problem.

186: Restore-heap range error.

*Program called standard procedure RELEASE, but the argument would attempt to restore the top of heap pointer to some value which is outside of the range of the current actual heap.

187: Stack overflow.

*Not enough local and/or estimated stack was allocated to run the program. This error typically means that the compiler's “best guess” of run-time stack memory requirements is wrong or that you have given overrides for the compiler generated values which are insufficient. The compiler cannot foresee the actual runtime routine calling structure and recursion activity, but its guess is in very many cases adequate. You may, however, from time to time need to use run time option overrides using the “`l`” option to override the local stack requirement assignment and the “`e`” option to override the estimated or working stack requirement. The required values can be deduced from the procedure table information which is printed at the end of each compilation if you know the actual nature of the run time routine activity, otherwise a “best guess” can be used and the run time option “`i`” can be invoked to get actual program statistics which show your override allocation, the actual needs of the program up to its termination point, and the amount of memory free that was overallocated.

188: MOD error, right argument is negative.

*Program used the MOD operator and has selected or defaulted to use the ISO version of the MOD function which cannot have a negative right argument. If a negative right argument is needed insert a “ISOMODE(false);” statement before the use of the MOD operator.

189: Pointer variable is expected.

*Scanning a call to standard procedure NEW and first parameter is not pointer-type.

*Scanning a call to standard procedure MARK or RELEASE and first parameter is not pointer-type.

190: Invalid PCODE instruction.

*The interpreter has detected a bad pcode instruction. Check system memory for errors then check the disk drive for errors. If the problem still persists, recompile the program to generate a new pcode file. If the problem still persists and you have not knowingly modified the pcode file, contact your nearest Radio Shack store.

191: Call-user-procedure range error.

*The interpreter has detected a bad reference to one of your procedures in the pcode file. Check the system as per error 190.

192: Pointer range error.

*Program is attempting to reference memory via a pointer, and the pointer does not point to a valid memory address. Check that the pointer has been properly initialized in your program. This error is caused by your program either not initializing the pointer correctly or setting it to an incorrect value via “trick” programming or non-standard programming.

193: Subscript or range error.

*An array is being indexed, and one of the index expressions is out of range for one of the array indices, or a value is being assigned to a subrange variable and the value is out of range.

194: Case error.

*A CASE statement is being executed, but the value of the case determinant does not exist as a selection anywhere for this statement, and the case selection OTHERWISE was not used.

195: Call-standard-procedure range error.

*An interpreter has detected an invalid reference to a standard procedure in the pcode file. Check system integrity as per error 190.

196* *

197: CHR function range error.

*Program is calling standard procedure CHR, but the value of the argument is not in the closed interval [0, 255] and thus cannot be mapped into a character.

198* *

199: Integer overflow on add, subtract, or negate.

200: Invalid Pascal option specification.

*A run time option specification is unrecognizable.

201: Pascal option has a value greater than 65535.

*The value for a run time option parameter is too big.

202: Local or Extended default stack requirement greater than 65535.

*The value for a run time option for “l” or “e” is too big.

203: OS-9 file error during processing of PCODE file, OS-9 error number follows.

*While attempting to open, close, or access the pcode file, an OS-9 error was detected. The program is aborted and the OS-9 error is displayed.

204: Invalid number of procedures in PCODE file, bad PCODE file.

*An interpreter has found an invalid pcode file header. Check system integrity as per error 190.

205: Compile-time errors detected in PCODE file, cannot run program.

*An attempt is being made to run a pcode file for which the compiler has detected errors. DO NOT ATTEMPT TO RUN THIS PCODE FILE! as in all likelihood it could crash your system.

206: Not enough memory to run program.

*There is not enough memory available at this time on your system to meet the needs for local, estimated, global, and heap requirements. Check to see if memory is fragmented. If running in a multi-tasking environment, it may be that other task activity is temporarily using memory which may be available later which might then allow you to run. If possible, use or adjust the run time option overrides to adjust the runtime memory requirements.

207: Ran out of swap buffers.

*While running a Pascal program via one of the interpreter products, there were not enough swap buffers allocated. This situation occurs when a routine is currently running which requires several buffers to be locked since they contain string constant information which may be needed by a buffer containing program code. As a result, the code swapper finds that another swap buffer is required for processing, but all available buffers are locked. To cure this, either allocate more swap buffers when running the program, run the program via a non-swapping interpreter, or rewrite the program so that fewer and/or smaller string constants are used in the routine which caused the fault.

208: Floating point overflow.

209: Floating point divide by zero.

210: Expected a boolean expression.

*Scanning a call to one of the following standard procedures or functions which requires a boolean expression for this parameter: IOABORT, ISOMOD, RIGHTJUST, MATHABORT.

*Scanning an IF statement and did not find a boolean expression following IF.

*Scanning a REPEAT statement and did not find a boolean expression following UNTIL.

211: Cannot issue CLOSE to SYSERR file.

*Scanning a call the standard procedure CLOSE, and the file name being referenced is the SYSERR file. This file must never be closed as it is used as the path for reporting run time errors.

212: A text file is not allowed here.

*Scanning a call to one of the following standard procedures which cannot operate on a text-type file: REPOSITION, POSITION, or FILESIZE.

213: Must be a 32 element character array.

*Scanning a call to the standard procedure GETINFO or PUTINFO, and the second parameter is not a 32 element character array.

214: Procedure or function names may not be passed as parameters (compiler restriction).

*An attempt is being made to pass a procedure or function by name. This compiler only allows functions to be passed by value.

215: Integer type variable name expected.

*Scanning a standard procedure call , and the parameter scanned is not the name of an integer which is to contain the result from this call.

216: Must be type real or integer.

*Scanning a call to the standard procedure ABS or SQR, and the argument is not integer-type or real-type.

217: Must be type real.

*Scanning a call to the standard procedure REPOSITION, and the second parameter is not real-type.

*Scanning a call to the standard procedure TRUNC or ROUND, and the argument is not real-type.

218: Scalar expected.

*Scanning a call to the standard procedure PRED or SUCC, and the argument is not scalar-type.

*Scanning a FOR statement and found FOR <ident>:=<expr> and possible TO/DOWNTO <expr> but <expr> is not scalar-type.

219: GOTO's may not lead out of enclosing proc/func (compiler restriction).

*Scanning a GOTO statement, and the label found is located in a different block than that containing the GOTO.

220: Square root of negative number.

221: Range error on floating point fix.

*Program is explicitly or implicitly attempting to fix a floating point number, and it can't fit in the 16 or 32 bit integer destination without loss of significance.

222: OS-9 error during processing of EXTERNAL tables, OS-9 error number follows.

*During program initialization, an EXTERNAL procedure could not be loaded because the module does not exist as specified in the external tables, or the external tables are not properly formatted.

**Use the PASCALE utility to properly format the external tables.

223: Nonexistent procedure called from an EXTERNAL procedure.

*During program execution, an EXTERNAL procedure attempted to call a procedure which does not exist in the current program.

224: Multiple decimal points encountered during CNVTREAL processing.

*Program called the standard procedure CNVTREAL to convert a string of characters into a real number, but two decimal points were found. See error 72 for further information.

225: Digit expected during CNVTREAL processing.

*Program called the standard procedure CNVTREAL to convert a string of characters into a real number, but at some point a digit was required and not found. See error 73 for further information.

226: Floating point overflow during CNVTREAL processing.

*Program called the standard procedure CNVTREAL to convert a string of characters into a real number, but the number is too large. See error 74 for further information.

227: Pcode file has been altered! Cannot run program.

*The pcode file that you are trying to run with one of the OS-9 Pascal interpreter products is either not a valid pcode file or has been altered in some way since the compiler produced the file. If you have simply given the wrong file name, just reenter the command with the correct pcode file name. If, however, the pcode file has been altered then do not run the file. The OS-9 Pascal program products try very hard to ensure that the pcode file has not been altered before they will process the file. If you attempt to circumvent this safety feature, it is quite likely that the integrity of the running system will be destroyed - usually with catastrophic results.

228: OS-9 error on restore heap call, OS-9 error number follows.

*Program called standard procedure RELEASE and OS-9 detected an error. Program aborts and OS-9 error is displayed.

229: LN of a zero number.

230: Must be a text file type.

*The following standard procedures can only operate on text files: READ, READLN, WRITE, WRITELN, INTERACTIVE, PAGE, OVERPRINT, SYSREPORT, and PROMPT.

231: User scalar types not implemented for reading or writing.

*Attempt to read or write user scalar types using one of the following standard procedures: READ, READLN, WRITE, or WRITELN.

232: Set has too many elements (compiler restriction).

*Scanning a set declaration, found a definition which requires a lower ordinal limit of less than zero or a higher ordinal limit of greater than 255. This version of the compiler allows sets of up to 256 elements.

233: Too many procedures declared (compiler restriction).

*An attempt is being made to define more than 254 procedures within a program block.

234: Integer value expected.

*An array reference using the <expression> FOR <count> language extensions is scanned but <expression> is not an integer type.

235* *

236: Type indicator is not defined for this object.

*Trying to determine the object type for a load or store pcode instruction, and a file-type, tagfield-type, or variant-type was found which cannot be handled.

237: variant or tagfield is not allowed here.

*The identifier scanned is either a variant-type or tagfield-type, neither of which is allowed here.

238: Period, BEGIN, PROCEDURE, OR FUNCTION expected.

*After having scanned a function or procedure body a period (end of source program), PROCEDURE or FUNCTION (define another routine), or BEGIN (begin an outer block) is expected.

239: LN of a negative number.

240: EXP of a number greater than 88.0296919.

241: SIN of a number greater than 102942.13 or less than minus 102942.13.

Appendix B. Pascal Syntax

The following information describes the syntax of the Pascal language. The “Backus-Naur Formalism” will be used to describe the correct structure of Pascal statements. The following symbols are meta-symbols, symbols that describe other symbols, belonging to BNF, but they are not part of the Pascal language.

{ } - Curly brackets indicate possible repetition of the enclosed symbols.

| - Or

[1]	program	: : =	program-heading block "."
[2]	program-heading	: : =	PROGRAM identifier "(" file-identifier { "," file-identifier } ")" ";"
[3]	file-identifier	: : =	identifier
[4]	identifier	: : =	letter { letter-or-digit | "_" }
[5]	letter-or-digit	: : =	letter | digit
[6]	block	: : =	label-declaration-part constant-definition-part type-definition-part variable-declaration-part procedure-and-function-declaration-part statement-part
[7]	label-declaration-part	: : =	empty | LABEL label { "," label} ";"
[8]	label	: : =	unsigned-integer
[9]	constant-definition-part	: : =	empty | CONST constant-definition { ";" constant-definition} ";"
[10]	constant-definition	: : =	identifier = constant
[11]	constant	: : =	unsigned-number | sign unsigned-number | constant-identifier | sign constant-identifier | string
[12]	unsigned-number	: : =	unsigned-integer | unsigned-real | unsigned-hexinteger
[13]	unsigned-integer	: : =	digit { "_" digit | digit}
[14]	unsigned-hexinteger	: : =	"$" hexdigit { "_" hexdigit | hexdigit }
[15]	unsigned-real	: : =	unsigned-integer . digit { digit} | unsigned-integer . digit { digit} E scale-factor | unsigned-integer E scale-factor
[16]	scale-factor	: : =	unsigned-integer | sign unsigned-integer
[17]	sign	: : =	"+" | "-"
[18]	constant-identifier	: : =	identifier
[19]	string	: : =	""" character { character} """
[20]	type-definition-part	: : =	empty | TYPE type-definition {";" type-definition} ";"
[21]	type-definition	: : =	identifier = type
[22]	type	: : =	simple-type | structured-type | pointer-type
[23]	simple-type	: : =	scalar-type | subrange-type | type-identifier
[24]	scalar-type	: : =	(identifier { "," identifier})
[25]	subrange-type	: : =	constant ".." constant
[26]	type-identifier	: : =	identifier
[27]	structured-type	: : =	unpacked-structured-type | PACKED unpacked-structured-type
[28]	unpacked-structured-type	: : =	array-type | record-type | set-type | file-type
[29]	array-type	: : =	ARRAY "[" index-type { "," index-type} "]" OF component-type
[30]	index-type	: : =	simple-type
[31]	component-type	: : =	type
[32]	record-type	: : =	RECORD field-list END

[33]	field-list	: : =	fixed-part | fixed-part ";" variant-part | variant-part
[34]	fixed-part	: : =	record-section { ";" record-section}
[35]	record-section	: : =	field-identifier { "," field-identifier} ":" type | empty
[36]	variant-part	: : =	CASE tag-field type-identifier OF variant { ";" variant}
[37]	tag-field	: : =	field-identifier ":" | empty
[38]	variant	: : =	case-label-list ":" "(" field-list ")" | empty
[39]	case-label	: : =	constant
[40]	set-type	: : =	SET OF base-type
[41]	base-type	: : =	simple-type
[42]	file-type	: : =	FILE OF type
[43]	pointer-type	: : =	"^" type-identifier
[44]	variable-declaration-part	: : =	empty | VAR variable-declaration { ";" variable-declaration} ";"
[45]	variable-declaration	: : =	identifier { "," identifier} ":" type
[46]	procedure-and-function-declaration-part	: : =	{ procedure-or-function-declaration ";"}
[47]	procedure-or-function-declaration	: : =	procedure-declaration | function-declaration
[48]	procedure-declaration	: : =	procedure-heading block
[49]	procedure-heading	: : =	PROCEDURE identifier ";" | PROCEDURE identifier (formal-parameter-section { ";" formal-parameter-section}) ";"
[50]	formal-parameter-section	: : =	parameter-group | VAR parameter-group
[51]	parameter-group	: : =	identifier { "," identifier} ":" type-identifier
[52]	function-declaration	: : =	function-heading block
[53]	function-heading	: : =	FUNCTION identifier ":" result-type ";" | FUNCTION identifier "(" formal-parameter-section {";" formal-parameter-section} ")" ":" result-type ";"
[54]	result-type	: : =	type-identifier
[55]	statement-part	: : =	compound-statement
[56]	statement	: : =	unlabelled-statement | label ":" unlabelled-statement
[57]	unlabelled-statement	: : =	simple-statement | structured-statement
[58]	simple-statement	: : =	assignment-statement | procedure-statement | go-to-statement | empty-statement
[59]	assignment-statement	: : =	variable := expression | function-identifier := expression
[60]	variable	: : =	entire-variable | component-variable | referenced-variable
[61]	entire-variable	: : =	variable-identifier
[62]	variable-identifier	: : =	identifier
[63]	component-variable	: : =	indexed-variable | field-designator | file-buffer
[64]	indexed-variable	: : =	array-variable "[" expression {"," expression} "]"
[65]	array-variable	: : =	variable
[66]	field-designator	: : =	record-variable . field-identifier
[67]	record-variable	: : =	variable
[68]	field-identifier	: : =	identifier
[69]	file-buffer	: : =	file-variable "^"
[70]	file-variable	: : =	variable

[71]	referenced-variable	: : =	pointer-variable "^"
[72]	pointer-variable	: : =	variable
[73]	expression	: : =	simple-expression | simple-expression relational-operator simple-expression
[74]	relational-operator	: : =	"=" | "<>" | "<" | "<=" | ">=" | ">" | IN
[75]	simple-expression	: : =	term | sign term | simple-expression adding-operator term
[76]	adding-operator	: : =	"+" | "-" | OR | "|" | "#"
[77]	term	: : =	factor | term multiplying-operator factor
[78]	multiplying-operator	: : =	"*" | "/" | DIV | MOD | AND | "&"
[79]	factor	: : =	variable | unsigned-constant | (expression) function-designator | set | NOT factor
[80]	unsigned-constant	: : =	unsigned-number | string | constant-identifier | NIL
[81]	function-designator	: : =	function-identifier | function-identifier (actual-parameter { "," actual-parameter})
[82]	function-identifier	: : =	identifier
[83]	set	: : =	"[" element-list "]"
[84]	element-list	: : =	element { "," element } | empty
[85]	element	: : =	expression | expression | expression ".." expression
[86]	procedure-statement	: : =	procedure-identifier | procedure-identifier (actual-parameter { "," actual-parameter})
[87]	procedure-identifier	: : =	identifier
[88]	actual-parameter	: : =	expression | variable | procedure-identifier | function-identifier
[89]	go-to-statement	: : =	GOTO label
[90]	empty-statement	: : =	empty
[91]	empty	: : =	
[92]	structured-statement	: : =	compound-statement | conditional-statement | repetitive-statement | with-statement
[93]	compound-statement	: : =	BEGIN statement { ";" statement} END
[94]	conditional-statement	: : =	if-statement | case-statement
[95]	if-statement	: : =	IF expression THEN statement | IF expression THEN statement ELSE statement
[96]	case-statement	: : =	CASE expression OF case-list-element { ";" case-list-element} END
[97]	case-list-element	: : =	case-label-list ":" statement | empty
[98]	case-label-list	: : =	case-label { "," case-label } | OTHERWISE
[99]	repetitive-statement	: : =	while-statement | repeat-statement | for-statement
[100]	while-statement	: : =	WHILE expression DO statement
[101]	repeat-statement	: : =	REPEAT statement { ";" statement} UNTIL expression
[102]	for-statement	: : =	FOR control-variable := for-list DO statement
[103]	for-list	: : =	initial-value TO final-value | initial-value DOWNTO final-value
[104]	control-variable	: : =	identifier
[105]	initial-value	: : =	expression
[106]	final-value	: : =	expression
[107]	with-statement	: : =	WITH record-variable-list DO statement

[108]	record-variable-list	::=	record-variable { "," record-variable}
[109]	letter	::=	"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
[110]	hexdigit	::=	digit | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f"
[111]	digit	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
[112]	character	::=	Any-ASCII-character-except-quote | """"

Appendix C. Quick Reference

Table C.1. Pascal Keywords

AND	END	MOD	SET
ARRAY	FILE	NIL	THEM
BEGIN	FOR	NOT	TO
CASE	FUNCTION	OF	TYPE
CONST	GOTO	OR	UNTIL
DIV	IF	OTHERWISE	VAR
DO	IN	PROCEDURE	WHILE
ELSE	LABEL	PROGRAM	WITH

Table C.2. Standard Procedures

FIELDPUT(variable-name, start-bit, length, value:integer)
ISOMOD(logical-value:boolean)
MARK(variable-name:pointer-type)
RELEASE(variable-name:pointer-type)
NEW(variable-name:pointer-type)
MATHABORT(logical-value:boolean)
RIGHTJUST(logical-value:boolean)
SYSTIME(year, month, day, hour, minute, second:integer)

Table C.3. Standard Functions

ABS(expression:integer-or-real):same-type-as-argument
ADDRESS(variable-reference):integer
AFRAC(expression:real):real
AINT(expression:real) :real
ARCTAN(expression:integer-or-real):real
CHR(expression:integer): char
CNVTREAL(string-or-char-array):real
COS(expression:integer-or-real):real
EXP(expression:integer-or-real):real
FIELDGET(expression, start-bit, length:integer):integer
LN(expression:integer-or-real):real
MATHRESULT: integer
ODD(expression:integer):boolean
ORD(ordinal-type-value):integer
PRED(ordinal-type-value):same-type-as-argument
ROUND(expression:real)::integer
SHELL(string-or~char-array):integer
SIN(expression:integer-or-real):real
SIZEOF(variable-or-type-name):integer
SQR(expression:integer-or-real):same-type-as-argument

SQRT(expression:integer-or-real):real

SUCC(ordinal-type-value>):same-type-as-argument

TRUNC(expression:real):integer

Table C.4. Standard I/O Procedures

APPEND(text-filename {,external-filename {,open-mode}})

CLOSE(filename)

GET(file-variable)

PUT(file-variable)

GETINFO(filename, 32-byte-structure)

PUTINFO(filename, 32-byte-structure)

IOABORT(filename, logical-value)

OVERPRINT(text-filename)

PAGE(text-filename)

PROMPT(text-filename)

READ({file-variable,} read-parameter-list)

READLN({text-filename,} read-parameter-list)

REPOSITION(file-variable, record-number)

RESET(file-variable {,external-filename {,open-mode}})

REWRITE(file-variable {,external-filename {,open-mode}})

SEEKEOF(file-variable)

SHORTIO(file-variable, record-length)

SYSREPORT({text-filename,} integer-value)

UPDATE(file-variable {,external-filename {,open-mode}})

WRITE({file-variable,} write-parameter-list)

WRITEEOF(file-variable)

WRITELN({text-filename,} write-parameter-list)

Table C.5. Standard I/O Functions

EOF({filename}):boolean

EOLN({filename}):boolean

FILESIZE(file-variable):real

GETCHAR(filename):char

INTERACTIVE(text-filename):boolean

IOREADY(filename):boolean

IORESULT(filename):integer

LINELENGTH(filename):integer

OPENED(filename):boolean

POSITION(file-variable):real

Table C.6. Pascal Operators

Arithmetic Operators	**Boolean Operators**
+ (unary), - (unary)	NOT
*, /, &, DIV, MOD	OR

Arithmetic Operators

+, -, !, #

Relational Operators

=, <>, <, >, <=, >=, IN

Boolean Operators

AND

Set Operators

*, +, -