ISO PASCAL

REFERENCE
MANUAL

ISO PASCAL

PART NO 0410, 010
ISSUE NO 1
JULY 1985

The product described in this manual and products for use with it, are subject to continuous developments and improvement. All information of a technical nature and particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

In case of difficulty please contact your supplier. Deficiencies in software and documentation should be notified in writing, using the Acorn Scientific Fault Report Form to the following address:

Sales Department
Scientific Division
Acorn Computers Ltd
Fulbourn Road
Cherry Hinton
Cambridge
CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised agents. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Published by Acorn Computers Limited,
Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.

Within this publication the term BBC is used as an abbreviation for the British Broadcasting Corporation.

NOTE: A User Registration Card is supplied with the hardware. It is in your interest to complete and return the card. Please notify Acorn Scientific at the above address if this card is missing.

ISBN 0 907876 38 2 Acorn Scientific

Contents

1 Introduction to ISO Pascal

1.1 The Pascal language

The Pascal programming language is a versatile, well-structured, and strongly-typed high level language originally developed for use as a teaching language, but now used for a wide variety of system and applications software.

The major features of Pascal are its ease of use, legibility, rigorous checking during compilation to reduce the number of coding errors and source portability.

Note that this document is a reference manual intended as a guide to the Acorn 32000 implementation of ISO Pascal; it is not a tutorial. Throughout this document, ISO Pascal refers to Pascal as implemented on Acorn 32000 based products and running under the Panos Operating System.

1.2 Using ISO Pascal

ISO Pascal is a compiled language. It consists of a two-pass compiler which translates Pascal source programs into 32000 machine code, and a collection of pre-compiled modules (the Pascal library) to provide facilities such as string to numeric conversion.

The first pass of the compiler checks that the program conforms to the rules of Pascal and the second pass constructs the equivalent machine code program.

In addition, useful facilities such as the ability to call routines written in other languages (such as C and FORTRAN 77) have been included. This uses the Acorn cross-calling standard introduced for use with Pascal in chapter 6 of this document and described in full in the *Panos Technical Reference Manual.*

The Pascal compiler produces extensive debugging and error trapping information, though this can be disabled using compiler options. There are also various extensions to the standard, e.g. bit-wise logical operations.

1.2.1 Installation

The Pascal compiler is supplied on a 5¼ inch double-sided floppy disc. Procedures for installation are shown in the relevant chapter in the appropriate *User Guide* supplied with this system. If the correct procedures are not followed, then Pascal may not function correctly. Note that even though Pascal has been supplied on floppy disc, and is going to be used with the DFS, it still must be installed. If you wish to know more about file extensions or utilities in general, consult the *Panos Guide to Operations.*

2 Using the compiler

There are a number of arguments which can be issued to the compiler which give extra control over the input and output of the compilation, and allow the compilation options to be used.

-list

This command specifies that a line-numbered listing is sent to 'Prog-lis' (by default), or a given file name.
For example: `Pascal PProg -list Pprogn` will place the listing in Pprogn-lis. The listing consists of the original source program along with numbered lines, and any messages that may have been generated during compilation. This can provide a useful reference when de-bugging. See figure 1 for a demonstration of this command. The numbering scheme is very simple: each physical line in a source file is numbered sequentially starting from one. Some line numbers are followed by the characters '{' or '+'. These constitute aids to tracking down potential sources of errors.
The symbol '{' signifies that the line is a continuation of a comment started on a previous line. Omitting the closing right brace '}' from a comment is a common mistake resulting in subsequent statements being taken as part of the comment. An 'extended comment' will be terminated by the next properly specified comment, leading to parts of the program disappearing mysteriously.
The '+' character indicates that the line is a continuation of the previous line, and pin-points places where semicolons might have been omitted.

-aof

The file containing the machine code equivalent program in Acorn Object Format, is automatically sent to a file with the same name as that of the source file, but with the extension '-aof'. This command allows the user to specify the name of the Acorn Object Format file. For example: `Pascal Prog -aof ObjProg` will place the object program in a file called 'ObjProg-aof'. Unless an alternative extension is given, the default '-aof' is added by the compiler.

-extend

This option makes the compiler accept programs containing any of the extensions to standard Pascal. By default, the compiler will not

recognise any of the extensions and use of them will result in compile-time error messages.

-nowarn
: This option prevents the compiler from issuing warning messages. As warning messages can often indicate trouble spots in a program, this option should only be used by programmers experienced in the use of the compiler.

-notify
: If the '-extend' option is used and '-notify' is also given, then

```
Warning 158 -- this is a non-standard construction
```

: is issued each time an extension is found.

-nochecks
: This option should only be used with programs that have already been fully tested. It prevents the compiler from generating code to

i. detect the use of unassigned variables while the program is executing;
ii. detect overflow during the evaluation of expressions;
iii. check that enough memory is available for the correct execution of the program.

-noDiags
: This option inhibits the generation of, i.e. information used in the event of program failure which locates the faulty source line by number and displays the names and values of active variables. Other than the behaviour following execution errors, this option has no effect on the execution of a program; however, it does reduce the size of the object file.

-identify
: This option requests identification of the compiler.

-help
: This option prints out help text.

Example compiler commands

A. The Minimal Command

```
Pascal Prog1
```

This command will compile the source program 'Prog1-Pas'; default settings are used throughout, i.e. the aof file is called 'Prog1-aof', no listing is generated, error messages are sent to the screen, warning messages are given, full checking and diagnostic tables are included in the object code, and Pascal extensions are unacceptable.

B. Specifying an error file and inhibiting diagnostic tables

```
Pascal dfs::2.Prog1 -error dfs::0.errs -nodiags
```

Compilation is carried out as in the first example, except that error messages are sent to the file 'dfs::0.errs', and no diagnostic tables are included in the object code.

3 A Description of ISO Pascal

3.1 Language reference

The language reference followed for ISO Pascal is the ISO 7185 Standard, available in a slightly revised form as BS 6192:1982 *Specification for Computer programming language Pascal*, from the British Standards Institution, and reprinted elsewhere.

3.2 Restrictions and variations

3.2.1 Implementation-defined features

This section describes those details of the language implementation which are left as implementation-defined ("possibly differing between processors but defined for any particular processor"), by the ISO 7185 Standard. The numbers in brackets below refer to sections in that standard.

(6.1.7)
String-characters may be any ASCII character. NL (ASCII 10) is currently excluded pending clarification of various statements made in the standard validation suite.

(6.4.2.2-b)
The real-type corresponds to the 64-bit double-precision floating-point numbers of the proposed IEEE standard for Binary Floating Point Arithmetic (Task P754).

(6.4.2.2-d)
The char-type defines values corresponding to the ASCII 7-bit character set, extended to include the ordinal values 128 .. 255.

(6.6.5.2)
Except in the case of text files the operations rewrite, put, reset and get are performed at the time the corresponding statement is executed. For text files the operation is delayed until the next actual or implied use of the buffer variable associated with the file.

(6.7.2.2)
MAXINT has the value 2147483647 (16_7FFFFFFF).

(6.7.2.2)
Real operations are performed to the accuracy of the proposed IEEE standard for Binary Floating Point Arithmetic (Task P754). Floating-point functions return results correct to 12 decimal places.

(6.9.3.1)
The default values for TotalWidth are:

Integer-type : 1
Real-type : 3
Boolean-type : 5

(6.9.3.4.1)
ExpDigits has the value 3.

(6.9.3.4.1)
The exponent character is 'e'.

(6.9.3.5)
The value TRUE is output as 'True' and the value FALSE is output as 'False', in other words for both values all letters are in lower case except for the first which is in upper case.

(6.9.5)
PAGE causes a form-feed character (ASCII 12) to be sent to the specified file.

(6.10)
Program parameters may only denote files, and these may only be of types which do not include pointers. Program parameters are bound to external files which correspond to the newline-delimited text files of the host system.

(6.1.9)
The following table gives the alternative tokens accepted by the compiler:

Reference token	Alternative token
↑	\
[	(.
]	.)
{	(*
}	*)

3.2.2 Implementation-dependent features

This section describes those details of the language implementation which are left as implementation-dependent ("possibly differing between processors and not necessarily defined for any particular processor") by the ISO 7185 Standard. The numbers in brackets below refer to sections in that standard.

(6.5.3.2)
The index-expressions of indexed variables are evaluated from left to right.

(6.7.1)
The expressions of a member designator are evaluated from left to right.

(6.7.1)
The member-designators of a set constructor are evaluated from left to right.

(6.7.2.1)
The operands of a dyadic operator are evaluated from left to right.

(6.7.3)
The actual parameters of a function-designator are evaluated from left to right.

(6.8.2.2)
The variable of an assignment statement is accessed before the expression is evaluated.

(6.7.3)
The actual parameters of a procedure-statement are evaluated from left to right.

(6.9.5)
Inspecting a textfile to which PAGE was applied will return the ASCII character Form File (code 12). This does not mark the end of a line.

(6.10)
Program parameters may only be bound to external entities which are files.

(6.6.5.2)
The file parameters to READ and WRITE are evaluated the number of times given by:

MAX(1), number of parameters involving function calls

(6.6.5.4)
The array parameters to PACK and UNPACK are evaluated once each for each implied assignment.

3.2.3 Data sizes

Integer	4 bytes, doubleword aligned
Real	8 bytes, doubleword aligned
Char	1 byte, byte aligned
Boolean	1 byte, word aligned
Set	32 bytes, doubleword aligned
Records	n bytes, doubleword aligned
Arrays	(first element), doubleword aligned
Enumerated	
1 .. 255 items	1 byte, byte aligned
256 .. 32767 items	2 bytes, word aligned
otherwise	4 bytes, doubleword aligned
Subrange (Ordinal value)	
0 .. 255	1 byte, byte aligned
-32768 .. 32767	2 bytes, word aligned
otherwise	4 bytes, doubleword aligned

3.3 Extensions to ISO 7185

Important: these extensions (with respect to ISO 7185) will only be accepted if the compiler is run with the -extend option.

3.3.1 Identifiers

Dollar ($) and underline (_) may be used as the second or subsequent characters in identifiers.

3.3.2 Non-decimal constants

The form BASE_DIGITS may be used to specify constants with radix other than ten. BASE is the decimal specification of the radix, and DIGITS is a sequence of letters or decimal digits where the letters A, B, C .... represent the values 10, 11, 12 ..... For example:

```
16_CAFE3 = 831459 = 8_3127743
```

3.3.3 Bit-vector operators

The following operators act on fullword integer values:

&	logical AND
|	inclusive OR
||	exclusive OR
<<	logical left shift
>>	logical right shift
~	one's complement

The precedence of these operators is as follows:

&, <<, >>	same as *
|, ||	same as +
~	same as unary -

3.3.4 Reset and rewrite

The file procedures 'reset' and 'rewrite' are extended to enable file variables to be linked with external (i.e. filing system) names. The first parameter is

the same as in the standard procedures. The second parameter is a string giving the name to be used by the current filing system. An example is:

```
reset (infile, ':1.text1');
rewrite (outfile, 'text2');
```

The strings may be variables, e.g.

```
reset (data,usersName);
```

A special case of the extended forms for 'reset' and 'rewrite' is the null string parameter, or carriage-return, e.g. `'reset (file,'')'`. In this case, the file is bound to the console i.e. the screen for output and the keyboard for input. The file must be a text file. This is useful in enabling users to specify the console as a file, for example, so that text can be seen on the screen instead of being sent to a disc file which has to be examined later. Programs which process data until the 'eof' condition becomes true for a file can still work when the keyboard is used. The character [CTRL] - [D] used as input sets eof to true for the input file (or any file reset as the console).

3.3.5 Miscellaneous functions

LINENUMBER returns the number of the source line containing the reference to the function. Source lines are numbered starting from one for each file contributing to the compilation (see $INCLUDE).

ADDRESS(anyvar : anytype) returns the machine address of the variable given a parameter.

SIZE(anyvar : anytype) returns the number of bytes occupied by the variable given as parameter.

3.3.6 OTHERWISE

The reserved word OTHERWISE may be used to introduce the final clause of a CASE statement making that clause the action to be taken if no other clause in the case statement has been selected.

3.3.7 $INCLUDE

The $INCLUDE statement may be used to include source text from other files during the compilation. The form of the statement is:

```
$INCLUDE 'a-file-name'
```

The lines in each included file are numbered from one.

3.3.8 Machine-code

The compiler permits a form of assembly language to be included in program text. It should be noted at the outset that this facility is only intended as a last resort when other methods of achieving a result are totally inappropriate. No responsibility whatever is accepted for erroneous behaviour of programs containing machine-code statements.

The general form of the machine-code statement is:

```
*<mnemonic>_<operand list>;
```

where <mnemonic> is a 32000 standard assembly language instruction and <operand list> is a list of one or more operands separated by commas. For example:

```
*MOVD_1,X;
*SVC_12;
*ADDR_142(0),4;
```

Because Pascal uses integer values rather than identifiers for labels, destinations of branch instructions must be followed by a colon:

```
*BNE_99:;
```

Machine-code does not permit access to non-local variables.

The following examples demonstrate the various operand types:

```
*MOVD_#123456,2;
*MOVD_\1,%external(1)+4;
*MOVD_X,%TOS;
*MOVD_12(7)[4:%B],0(%SB);
*MOVD_12(16(%PC)), 16(12(%FP))
```

3.3.9 Modules

Modules provide a means for separate compilation. A module may contain definitions of procedures, functions and variables, some of which may be

made available to other programs or modules by preceding their definition with the reserved word EXPORT. In order to overcome limitations in linkage mechanisms and to give flexibility to naming conventions, an optional ALIAS may be given to a procedure or function identifier. This alias defines the string to be used for external linkage; it has no naming significance within the text of the program. If no alias is given the procedure or function name is used as the linkage name.

Procedures and functions which have been exported from other modules may be referenced by giving a procedure or function heading preceded by the reserved word IMPORT.

Variables global to the module must be prefixed by one of EXPORT, IMPORT or STATIC. The use of VAR for global variable declarations is not allowed in modules. Preceding variable declarations by EXPORT allows them to be accessed by other modules that contain similar variable declarations preceded by IMPORT.

Global variables preceded by STATIC remain local to that module but retain their values between calls to that module. There exists a mechanism to allow STATIC and EXPORT variables to have initial values assigned at their declaration.

For example:

```
TYPE colour = (red, green, blue);
EXPORT counter : integer := 0;
EXPORT a : ARRAY [1..6] OF integer := 12,7,23,5,1,5;

STATIC stream : integer;
STATIC called : boolean := false;
STATIC colourmap : ARRAY [2..5] OF colour := red, red, green, red;
```

The variable 'counter' is initialised to zero. The integer array 'a' is initialised to the six values following the variable declaration. Arrays cannot be partially initialised, i.e. all or none of the elements should be initialised. The variable 'stream' has no initial value but, once set, will hold its value between calls to this module. The boolean variable 'called' is initialised to the value 'false' and will have this value when the module is first entered. Neither 'stream' nor 'called' can be accessed from outside this module.

The whole module must start with the statement: MODULE module-identifier; and end with the statement END.

For example:

```
MODULE example;
   EXPORT counter : integer := 0;

   STATIC called : boolean := false;

   IMPORT size : integer;

   IMPORT FUNCTION other ALIAS 'EY_FN'(w : integer) : integer;

   FUNCTION setup(p : integer) : integer;  {a local function}
   BEGIN
      counter := counter+1;
      if called then setup := other(counter*size)
                else setup := other(p*size);
      called := true;
   END;

   EXPORT FUNCTION inner ALIAS 'EX_INNER'(p, q : integer) : integer;
   BEGIN
      inner := setup(p) + other(p*q*size);
   END;
END.  {of module}
```

In this example 'setup' is only accessible locally whereas 'inner' is available to other modules/programs via the linkage name 'EX_INNER'.

4 Compatibility

Programs developed for ISO Pascal compilers, for example, work without alteration on the 32000 system, providing no system dependent extensions have been used. This compatibility only applies to the source form, of course. A language developed using another ISO Pascal would have to be re-compiled before it could be run on Acorn Cambridge Series computers.

4.1 Compatibility with Acornsoft ISO Pascal

The following differences need to be taken into consideration when compiling programs written for Acornsoft Pascal with 32000 ISO Pascal. Both versions of Pascal conform to the ISO standard, however, there are a few differences in the extensions to the standard. These arise because the two compilers have different objectives. For example, one of 32000 Pascal's objectives has been to permit porting software from superminicomputers and mainframes, whereas Acornsoft Pascal has been constrained by limitations with space, and the need for compatibility with other BBC Microcomputer software.

4.1.1 Implementation-defined features in Acornsoft Pascal

The numbers in brackets refer to sections in the ISO 7185 standard.

(6.1.7)
The ASCII characters 4 (EOF) and 13 (CR) are excluded from string-characters.

(6.4.2.2-b)
The real-type corresponds to the 40-bit double-precision floating-point numbers of the proposed IEEE standard for Binary Floating Point Arithmetic (Task P754).

(6.7.2.2)
Real operations are performed to the accuracy of the proposed IEEE standard for Binary Floating Point Arithmetic (Task P754). Floating-point functions return results correct to 10 decimal places.

(6.9.3.1)
The default values for TotalWidth are:

Integer-type : 12
Real-type : 12
Boolean-type : 5

(6.9.3.4.1)
ExpDigits has the value 2.

(6.9.3.5)
The value TRUE is output as 'TRUE' and the value FALSE is output as 'FALSE'.

(6.9.5)
PAGE causes an eoln character, then form-feed (if required) to be sent to the specified file.

4.2 Data sizes in Acornsoft Pascal

These are the same as for 32000 ISO Pascal, with these exceptions: real data as 5 bytes, doubleword aligned, and enumerated data does not exceed 255 items. Packed data sizes are:

Subrange	
0 .. 255	1 byte, byte aligned
0 .. 65535	2 bytes, word aligned
any other	4 bytes, doubleword aligned

4.3 Limitations in Acornsoft Pascal

4.3.1 Identifiers

Only underline (_) may be used as the second or subsequent character in identifiers. Dollar ($) is not permitted.

4.3.2 Non-decimal constants

The only non-decimal constants in Acornsoft Pascal are hexadecimal (these are represented by &HexNumber).

4.3.3 Bit-vector operators

The bit-vector operators in Acorn 32000 ISO Pascal do not apply to Acornsoft Pascal. Note that '&' is used to represent hexadecimal values, and '~' is used to print these.

4.3.4 Miscellaneous functions

LINENUMBER and SIZE are not available in Acornsoft Pascal, although ADDRESS can be mimicked with a machine-code procedure.

4.3.5 OTHERWISE

In Acornsoft Pascal, the END statement precedes OTHERWISE:

```
CASE Thing OF
   item : stuff;
   blah : real;
END
OTHERWISE .....
```

4.3.6 INCLUDE

The INCLUDE statement does not exist in Acornsoft Pascal (a compiler option is used to compile other programs).

4.3.7 Machine-code

Assembly code is not permitted in Acornsoft Pascal programs. Procedures are used instead to call machine code.

4.3.8 Modules

There are no modules in Acornsoft Pascal.

4.3.9 Warning messages

No warning messages are given in Acornsoft Pascal.

4.4 Extensions in Acornsoft Pascal

The following procedures and functions are present in Acornsoft Pascal, but are not available in Acorn 32000 Pascal. They deal mainly with graphics and accessing facilities provided by BASIC. A functionally equivalent procedure library could be written to include most of these.

Procedures:

```
Mode, vdu, plot, point, sound, envelope
oscli, adval, inkey, settime.
```

Functions:

```
Time, lval, rval.
```

Shove procedures:

```
release
claim
free
```

5 Errors and debugging

5.1 Compile-time messages

5.1.1 Warning messages

Warning messages do not indicate that the program contains an error; they bring attention to various features of the program which suggest that something might be wrong. For example, one feature of Pascal which causes a lot of confusion to beginners is exactly where and when to use the semicolon. This can lead to semicolons being scattered liberally throughout the program resulting in statements like:

```
IF cases > maximum THEN;
    writeln('there were too many cases');
```

According to the rules of Pascal there is nothing wrong here, but it is unlikely that the effect will be what the user wanted. The semicolon following the THEN makes the IF control an empty (or null) statement. The 'writeln' will always be executed regardless of the values of 'cases' and 'maximum'. Once again it is important to stress that there is nothing wrong with the program if the effect described above is what was wanted, but as it is very likely to be a mistake the compiler will give the warning:

```
  IF cases > maximum THEN;
                         ↑
Warning 5 -- check that an empty statement is really wanted here
```

Below are descriptions of all of the warning messages produced by the ISO Pascal compiler.

```
Warning 4 -- this statement can never be reached
```

This warning is issued when the compiler detects that the indicated statement can never be executed. This is not an error, but frequently indicates an area in the program which should be checked carefully. It should be noted that this warning is only generated when it is patently obvious at compile-time that the statement cannot be executed;

non-occurrence of this message cannot be interpreted as meaning that statements can be executed.

```
PROGRAM warn(output);
CONST check = false;
BEGIN
   IF check THEN writeln('checking is enabled');
END.


Warning 5 -- check that an empty statement is really wanted here
```

Certain constructions in Pascal control the execution of a statement, e.g. IF and WHILE. In such cases it is possible to specify that a null or empty statement be controlled. This is acceptable Pascal, but the compiler will issue a warning as it is very likely that the null statement is the result of a misplaced semicolon.

```
PROGRAM warn(output);
   {this program will only output ONE line because of the}
   {dubious semicolon following the DO}
VAR n:1..10;
BEGIN
   FOR N := 1 TO 10 DO;
               writeln('Hello there');
END.


Warning 9 -- only the first 253 characters of this identifier
          -- will be used
```

This warning is issued when an identifier contains more than 253 characters. As it is not at all easy to construct a program containing identifiers of anything like this length, it is highly unlikely that the warning will ever be issued except from programs which deliberately set out to generate it.

```
Warning 19 -- <name> has not been used

Warning 19 -- the field <name> of the record at line <number>
           -- has not been used
```

This warning indicates that <name> has been declared but has never been used. It is intended to be used to help remove redundant variables and to

pinpoint some obscure errors. For example, it can sometimes indicate an identifier which has been spelled incorrectly resulting in a global variable being used by accident. The second form of the message is issued when a field of a record has never been mentioned explicitly.

```
PROGRAM warn;
VAR thing : integer;        {a global declaration}

PROCEDURE inner;
VAR thinl : integer;     {should have been 'thing'}
BEGIN
   thing := 0;           {accesses the global variable}
END;                     {thinl has not been used}

BEGIN
   inner;
END.


Warning 19 -- label <number> has not been used
```

The given label has been declared and used to mark a statement but has never been used as the destination of a GOTO statement. This is not an error but may well indicate a potential mistake.

```
PROGRAM warn(output);
LABEL 123;
VAR x : integer;
BEGIN
     x := 0;
123: writeln('complete');
     x := 1;
END.                        {label 123 has not been used}

Warning 28 -- check that label <number> really should be outside
           -- this block
```

The specified label has been referenced from a procedure or function declared inside the block which declared the label. This is correct Pascal but because integer values must be used for labels it is very easy to use the wrong label. The warning draws attention to places where an error in specifying a label could have an obscure effect on the program.

```
PROGRAM warn(output);
LABEL 1;
PROCEDURE jump; BEGIN
   GOTO 1;
END;

BEGIN
   jump;
1: writeln('here');
END.

Warning 85 -- this case label is out of range
```

Surprisingly, the Pascal standard does not proscribe the specification of case labels which are outside the range of the case selector except when the CASE is used to define record variants.

```
PROGRAM fail(output,input);
TYPE small = 1..3;
     rec   = RECORD CASE s:small OF
                      1: (a:integer);
                      2: (b:char);
                      3: (c:real);
                      4: (d:boolean); {error 4 is not in small}
             END;
VAR tiny:small;
BEGIN
   read(tiny);
   CASE tiny OF
      1: writeln('one');
      2: writeln('two');
      3: writeln('three');
      4: writeln('four');  {warning 4 is not in small}
   END;
END.

Warning 89 -- the case statement did not include all possible cases
```

A case statement has been found which has not provided a meaning for all the possible values of the case selector. When this occurs in a RECORD definition it is an error. When it occurs in a CASE statement it is only a warning, however if the case selector takes one of the unspecified values during execution a run-time error will be reported.

```
PROGRAM fail(output,input);
TYPE small = 1..3;
   rec  = RECORD CASE s:small OF
             1: (a:integer);
             3: (b:real);
   END;     {error - no case 2}
VAR tiny:small;
BEGIN
  read(tiny);
  CASE tiny OF
   1: writeln('one');
   3: writeln('three');
  END;             {warning - no case 2}
END.
```

```
Warning 158 -- this is a non-standard construction
```

This warning is issued if the compiler detects the use of an extension to standard Pascal and the -notify compile-time option has been selected.

```
Warning 161 -- this comment is within another comment
```

This warning is the result of the compiler discovering the opening bracket, '{' or '(*', of a comment while it is processing a comment. Although this is in no way an error it can point out problems caused by omitting or mistyping the closing bracket of a previous comment.

```
PROGRAM warn(output);
BEGIN
  {this comment is not terminated properly]

  writeln('this line will not be output');

  {this comment will terminate the comment started
  back on the line following BEGIN}
END.
```

```
Warning 162 -- this type only defines a single value
```

This warning is issued when a type definition results in an object which may only ever be given a unique value. This is not an error but is so strange that it is almost certainly the result of some mistake.

```
PROGRAM warn(output);

VAR unique : 12..12;  {very strange}
BEGIN
  unique := 12;     {the only value it can ever be given}
END.
```

5.1.2 Non-Fatal Error messages

As the compiler translates the program into machine code, it checks that, amongst other things, none of the rules of the Pascal language have been broken. If an error is detected, a message will be displayed giving the nature of the error and an indication of where it occurred. However, in contrast to Fatal errors (described in the next section), the compiler will not necessarily abandon processing because an error has been detected. As an example, if the program contained a statement declaring the same identifier twice, then the following message would appear:

```
20   VAR  tom, dick, harry, tom : integer;
          ↑

*ERROR 20 -- TOM has already been declared in this block
```

The first line shows the original program line in which the error was detected preceded by its line number, in this case 20. The compiler numbers each line in the program starting at the first line, so that errors may be located easily, either by examining a listing file or by using an editor.

The second line is blank except for the ↑ character which points into the line above, marking the position where the fault was detected. This line will be omitted from error messages in cases when it would give no useful information.

The third line is an explanation of the error. A brief description of some of the less obvious ones is given in Appendix A.

Figure 1 shows a compilation of a Pascal program from within the Panos editor. The source is in the background window, the compilation command in the upper command line window, and the listing file has been loaded into the lower window.

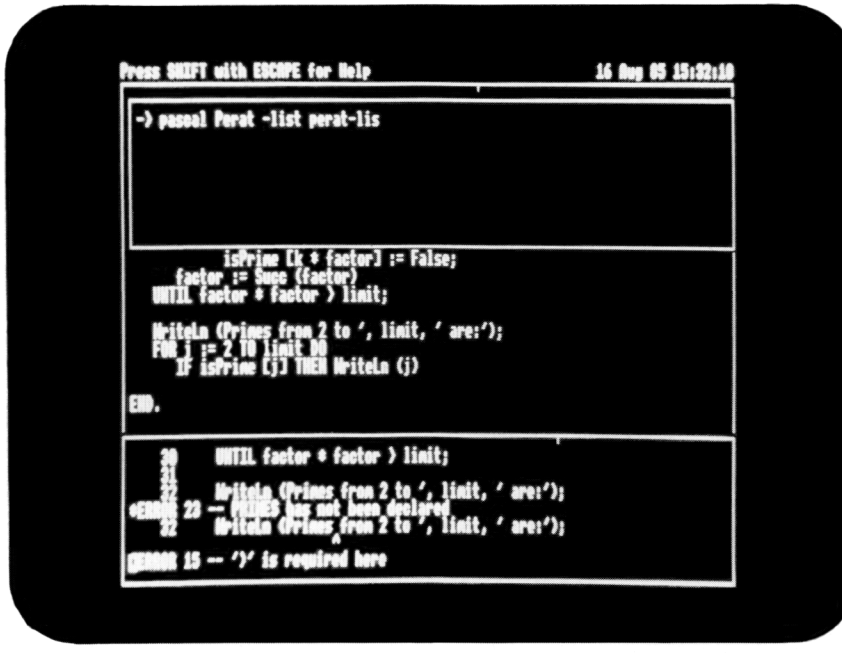


Figure 1 A Demonstration of a Pascal Compilation

Care should be exercised when interpreting error messages, as it is very common for the message to indicate a point in the program which is not the position of the actual error.

For example, the Acorn Cambridge Workstation and the BBC Microcomputer have the two symbols plus (+) and semicolon (;) on the same key (the SHIFT key being used to select one or the other). It is quite likely then that a plus could be mis-typed in place of a semicolon, giving rise to the following piece of program where the second plus should be a semicolon.

```
average := (this + that) / 2 +
product := this * that;
```

The compiler will see these two statements as a single statement:

```
average := (this + that) / 2 + product := this * that;
```

This is a valid statement up to the second := at which point the compiler will report an error:

```
product := this * that;
        ↑
*ERROR 11 -- a semicolon is required here
```

As you can see, the error is not reported at the exact point at which the mistake was made.
It is also common for errors in one part of a program to cause consequential errors in other parts. For example, consider the following program:

```
PROGRAM test(output);
VAR sum.difference:integer; {error . should be ,}
BEGIN
   sum := 12 + 7;
   difference := 12 - 7;
END.
```

Because there is an error in the declarations of 'sum' and 'difference', the compiler will report spurious errors when they are used. The actual errors generated by this program are:

```
2  VAR sum.difference:integer; {error . should be ,}
           ↑
*ERROR 15 -- ':' is required here

5      difference := 12 - 7;
       ↑
*ERROR 23 -- DIFFERENCE has not been declared
```

Note that even though the compiler has detected the error in line 2 at the correct place, it has made the wrong guess as to the cause. A statement of the form:

```
VAR sum:integer;
```

was expected.
If you cannot understand why an error has been reported on a particular statement, it may be that an error exists earlier in the program, and the error message is a result of a previous mistake.

Appendix A contains descriptions of error messages which may be produced by the compiler. Each description is accompanied by one or more complete (albeit incorrect) programs which demonstrate the fault, or the

correct use of features which the error indicates have been misused. In the error message statements, any item enclosed in angle brackets e.g. <something> will be replaced by an item appropriate to the context of the message.

5.1.3 Fatal compile errors

Certain errors cause the compiler to abort the compilation without trying to analyse the rest of the Pascal program.

Fatal errors are marked by the text

```
*FATAL ERROR nnn --
```

followed by the reason for stopping. The only way to prevent the same fatal error occuring again is to change the program in the way recommended. For example, if the fatal error was caused by the end of program token 'end.' being encountered too soon, look through the source for a '.' where there should be a ';'.

The range of FATAL errors produced by the compiler is listed in Appendix B.

5.2 Run-time errors

When an error is detected at run-time, an event is signalled to the diagnostic package which generates a post-mortem backtrace of the error.

Following the report of the nature of the error, information is displayed about the environment of the statement causing the error. This information includes the line number at which the error was detected, the name of the block containing that line, and the values of any local variables declared in that block.

If no value has yet been given to a variable, its value is printed as 'not assigned'. Large integer values will be printed in both decimal and hexadecimal (base 16) representations. If an integer value corresponds to the ASCII code for a printable character, then that character will be displayed as well. This process is repeated for the statement (usually a procedure call) which invoked this block, and so on back to the start of the program. Finally, the error and information about where it occurred is repeated in case the original copy has been scrolled off the top of the screen.

An example of a Pascal run-time error can be seen in figure 2. This shows the error message and backtrace produced when a type limit has been exceeded.

Figure 2 Example run-time error

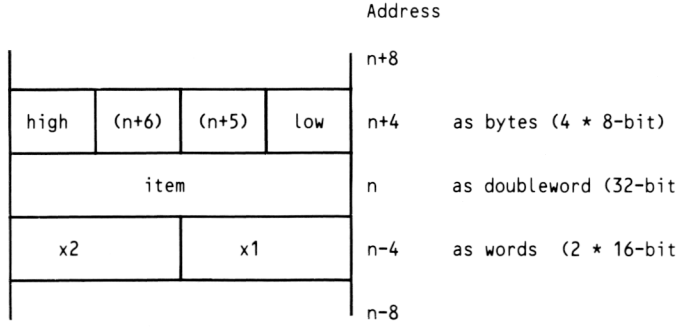
Appendix C contains a list of execution errors detected by Pascal. Note that some of these will go undetected if error checking options have been inhibited.

6 Using Pascal with other languages

6.1 Introduction

This chapter is a review of the outline code mechanisms of the Acorn 32000 Pascal system. It is intended to include sufficient detail to enable a program written in Pascal to call a procedure written in any high-level language which conforms to the Acorn Inter-Language Calling Standard (this includes the Panos interface procedures as documented in the *Panos Programmer's Reference Manual*), or in assembler. The Inter-Language Calling Standard is specified fully in the *Panos Technical Reference Manual*. A working knowledge of the 32000 architecture and instruction set is assumed, in explaining the low-level technical details. See section 6.4 for examples.

In the representation of memory in diagrams given here, numerically greater addresses are represented higher up the page than lower addresses. Bytes are arranged such that more significant (higher addressed) bytes appear to the left of less significant ones, where these are represented at the same height. Addresses where marked are given at the right-hand edge of a diagram, and correspond to the least significant byte of the doubleword referred to. This matches the natural byte ordering of the 32000-series architecture, and is shown by example below:



The form 'x x x x' will be used to represent memory whose contents are not defined for the purposes of an illustration.

6.2 Conformance

The Pascal implementation conforms very closely to the code mechanisms defined in the Inter-Language Calling Standard. In particular the way in which parameters are passed is exactly in accordance with the standard, as far as Integers, Reals, Records and Booleans are concerned. Similarly, results (such as Pascal allows, e.g. not Records) are returned in the standard manner. Since Pascal does not have a String type, inter-facing with languages which do (and with Panos) typically involves a little work in mapping strings onto a corresponding Pascal structure, but will normally be quite straightforward (i.e. there is no fundamental obstacle to doing this). Further, returning Record results is very simply mapped onto updating a VAR parameter. Full details of these mechanisms will now be given.

6.2.2 Types

Here is a full description of how Pascal types are mapped onto the architecture of the 32000 series:

Integer

32-bit quantity, aligned on 4-byte boundary. A variable of type Integer may hold any 32-bit signed value, except that when code is compiled with checks for assignment included (the default), the value 16_80808080 is reserved for the purpose of detecting use of a variable before it has been assigned to.

Real

64-bit double precision (32000 '-L' type), aligned on a 4-byte address boundary. See note below on the use of 32-bit reals. If assignment checks have been included in the compiled program then real variables are initialised with the reserved pattern 16_8080808080808080.

Char

8-bit unsigned byte quantity, without special alignment. When compiled with checking for assignment, Pascal code may not access character variables whose ordinal value is 16_80, since this is the value, preset into variables, which is tested for on accessing a Char, to detect use before assignment. With this check turned off, any 8-bit unsigned value (i.e. 0..255) may be assigned to and read from a Char variable.

Boolean

8-bit unsigned byte, without special alignment. The boolean values False and True are represented respectively by the binary values 0 and 1. As for type Char, if assignment checking is in force then the value 16_80 indicates that the variable has not been assigned to. The compiler, in generating code to test the state of a single Boolean variable, produces the instruction 'CMPQB 0, var', i.e. the variable is held to be True if its value is non-zero. Note however that this does not mean that ANY non-zero value will suffice to represent True, since for the operations AND, OR and NOT, the code generated relies on variables having the standard values if it is to execute correctly.

Enumerated

Variables of an enumerated type occupy memory according to the number of values defined by the type. For types with 1..256 values a single byte will be allocated for each variable. No particular alignment of the byte is imposed by the compiler. For types which define 257..32768 values a 16-bit word aligned on a 2-byte boundary is allocated, and in the unlikely event of more than 32768 values being defined by the type, a 4-byte doubleword, aligned on a 4-byte address boundary, would be allocated.

Subrange

Variables of subrange type occupy an amount of memory determined by the values of the subrange bounds. For types where the ordinal values of the lower and upper bounds both lie in the range 0..255, a single byte,

byte-aligned, is allocated for each variable; if they both lie in the range -32768..32767, a 2-byte word is allocated, aligned on a 2-byte boundary; otherwise a 4-byte doubleword is allocated, aligned on a 4-byte boundary.

Record

The structure of a record is a function of the types of its fields. In general a simple field in a record will be aligned as specified in the descriptions above, relative to the start of the record. The record as a whole will be aligned on a 4-byte boundary, and the total size of the record is always rounded up by the compiler to be a multiple of 4 bytes. A field of a record which is itself a record will follow the same rules (recursively). Note that the keyword PACKED has no effect at all (in the Pascal compiler described, Version 2.2.4), hence the structure and alignment requirements of a packed record are the same as for a non-packed one. Records having variant sections are implemented in such a fashion that the variant having the largest overall size determines the total allocated size of the record; this is true even when the heap allocation mechanism, NEW, is used with the tags of the variant selectors being provided. This latter point may change in a future version of the Pascal system, however.

Pointer

Pointers occupy 32 bits, aligned on a 4-byte address boundary. There are no unusual points to observe about their use, other than that when Pascal code is compiled with assignment checks, every access via a pointer is checked, to ensure that (a) the pointer is not NIL, and (b) that it points to a block of heap memory previously allocated by the Pascal NEW mechanism. In consequence of check (b), any Pascal code which interfaces to another language system, or Panos, where pointers are imported via the interface, should be compiled with these checks turned off; otherwise the program is liable to fail on check (b), since a foreign pointer will not in general satisfy this condition. A pointer is foreign if it occurs either as a parameter to an exported Pascal procedure or function, or as the result of an imported non-Pascal function.

Array

Arrays are implemented in the obvious simple fashion, as contiguous linear sequences of elements, each of which will be aligned as the element type requires. It should be noted that, as for records, use of the keyword PACKED has no effect on the storage layout adopted for arrays. Multi-dimensional arrays are stored in order such that the elements of an array accessed by the two sets of subscripts [i1,i2..ij] and [i1,i2..ij + 1] are stored in adjacent memory locations. This is a consequence of the definition of multi-dimensional arrays in Pascal, i.e. ARRAY [T1; T2] OF T3 is defined to be the same as ARRAY [T1] OF ARRAY [T2] OF T3;

Note

32-bit single precision reals are not used in Pascal, and so there is no simple way to interface to a procecdure with parameter(s) or result(s) which are 32-bit reals, unless such objects are not to be interpreted or modified by the Pascal code. In this case it is possible to treat the object simply as an integer, for purposes of assignment and parameter passing. However by using the facilities of extended Pascal to incorporate machine code in a program, such an object may be converted into a standard Pascal Real variable, and vice versa. This process is illustrated below:

```
TYPE
   Real32 = Integer;  {NB Cannot manipulate Real32's directly}

FUNCTION  RealOfReal32 (r: Real32): Real;
   VAR
      res: Real;
   BEGIN
      *MOVFL_r,res;        {MOVe Floating (32-bit) to Long (64-bit)}
      RealOfReal32 := res
   END;

FUNCTION  Real32OfReal (r: Real): Real32;
   VAR
      res: Real32;
   BEGIN
      { NB the instruction below will fail (i.e. generate an }
      {    exception) if r is too large to be represented as }
```

```
    {    a 32-bit real number }
    *MOVLF_r,res;  {MOVe Long (64-bit) to Floating (32-bit)}
    Real32OfReal := res
END;
```

6.2.3 Parameters

Parameters to a Pascal procedure are evaluated left-to-right, but are pushed on the stack (using the 32000 TOS mechanism) in the order defined by the Calling Standard, i.e. right-to-left. The following details govern the individual parameter mechanisms:

VAR parameters

For all parameter types except conformant arrays, the address of the object concerned is pushed as a 32-bit unsigned pointer; for (non- conformant) arrays this is the address of the first element of the array.

A conformant array is passed as a VAR parameter by means of a number of 32-bit items, pushed on the stack in the following order:

1. The address of element [0] (or [0,0] etc) of the actual array parameter. This is defined even if there is no such element in the array; for instance the address of the 0 element of an array specified as 'ARRAY [2..5] OF Char' is 2 bytes less than the address of the first actual element. Where the bounds are of an enumerated type, the ordinal value of the lower bound is used for this calculation.
2. For each dimension of the array, working from the right-most index to the left-most, 3 items are pushed:

    a. the ordinal value of the lower bound of this dimension of the actual array parameter;

    b. the ordinal value of the upper bound;

    c. the size in bytes of this dimension of the array: this is determined by considering the array in the form:

```
ARRAY [T1] OF ARRAY [T2] .. OF ARRAY [Tn] OF TE
```

For each dimension j (= 1..n), the dimension size is computed as the number of values in the type Tj multiplied by the size of the type to the right of the corresponding 'OF'; for dimension n this is the number of values in Tn multiplied by the size of the element type TE, for dimension k < n it is given by the expression:

size of dimension (k + 1) * number of values in type Tk

Hence the total number of 32-bit items pushed for a conformant array parameter is given by the expression:

```
1 + (number of dimensions) * 3
```

For example, when an actual array parameter specified as:

```
Jim: ARRAY [1..10, 0..3] OF Integer
```

(which could equally be specified as:

```
Jim: ARRAY [1..10] OF ARRAY [0..3] OF Integer)
```

is passed to a procedure specified as:

```
PROCEDURE Fred (VAR par: ARRAY [lo1..hi1; lo2..hi2] OF Integer)
```

the code generated is:

```
ADDR    Jim-16, TOS    ; 0th element 16 bytes before Jim[1,0]
MOVQD   0, TOS         ; low bound(2)
MOVQD   3, TOS         ; high bound(2)
ADDR    _16, TOS       ; dimension size(2)
MOVQD   1, TOS         ; low bound(1)
ADDR    _10, TOS       ; high bound(1)
ADDR    _160, TOS      ; dimension size(1) = total size of array
```

Value parameters

Integer, Char, Boolean, and Enumerated

Parameters of any of these types are pushed as 32-bit (doubleword) items. Where the basic storage size of a parameter type is less than 4 bytes, the object is zero-extended to occupy the full 32-bit parameter slot on the stack.

Subrange

Parameters of subrange types are pushed as 32-bit items, and if the lower bound of an integer subrange is negative then the object is sign-extended to 32-bits if necessary (ie. if the basic storage size is less than 4 bytes); all other parameters of subrange type are zero-extended if necessary.

Pointer

Parameters which are pointers to any type are pushed as 32-bit unsigned objects.

Real

Parameters of type Real are pushed as 64-bit (8-byte) floating- point quantities, using the instruction form 'MOVL <par>, TOS'.

Record

For Record types, the item which is pushed on the stack for a value parameter is the address of the actual record parameter (i.e. of a variable, since there are no record-type expressions in Pascal). It is the responsibility of the called procedure to copy the parameter value, via this pointer, into its own stack frame so that the original record is not modified by the called procedure (as required by Pascal semantics).

Non-conformant array

As for VAR parameters, the address of the first element of the actual array parameter is pushed. It is the responsibility of the called procedure to copy the array via this pointer into its own stack frame so that the original array is not modified. For a non-conformant array no size information need be passed since the size is fixed by the type of the array.

Conformant array

The information pushed on the stack for a value-type conformant array parameter is exactly the same as that for a conformant array parameter passed as VAR. It is the responsibility of the called procedure to copy the array into its own stack frame so as to prevent modification of the actual parameter. The size and bounds information passed is used to control this copying operation.

6.3 Interfacing to Panos standard procedures

To convert a procedure call specification given in abstract form (e.g. one from the *Panos Programmer's Reference Manual*) into a form suitable for use in a Pascal IMPORT statement, the following process should be applied. (Note that standard procedures having procedure-type parameters cannot be called from Pascal, and so should not be imported, and Pascal procedures with procedure-type parameters similarly cannot be called from a non-Pascal environment, and so should not be exported other than to another Pascal module. This is because procedures in the calling standard are passed purely by descriptor (they are not closures, i.e. there is no value environment in which they expect to operate), whereas Pascal procedure parameters are passed as full closures, and not in the standard manner.

1. Write down the abstract specification of the procedure.

2. If the type of any parameter or result includes the structure RECORD(<format>), write a suitable record type specification in Pascal for each such format. If the form 'RECORD(<format>) REF' occurs, also define a type '<rec_type>_ref' as '↑<rec_type>'. Rewrite the specification using these new type names instead of the original forms, and change the abstract syntax into the Pascal form, i.e. '<param_name>: <type>'.

3. If any parameters are of type STRING, then for each:

(a) decide on the maximum length of string which will be supplied to the procedure from this module, for that parameter;

(b) define a type String<N> (if it does not already exist), where <N> is the maximum length decided on in (a), and the type is defined as PACKED ARRAY [1..<N>] OF Char;

(c) rewrite the parameter specification as two items: the first having the original parameter name, but written in Pascal form, with its type being String<N>; the second having the parameter name with '_len' appended, of type Integer, i.e. 'STRING:<par_name>' is rewritten as '<par_name>: String<N>; <par_name>_len: Integer'.

4. If any parameters or results are of type HIDDEN, CARDINAL or INTEGER, rewrite these parameter specifications as the parameter name followed by the type Integer, e.g. 'HIDDEN:<par_name>' becomes '<par_name>: Integer', and so on.

5. For all results other than the first (left-most), move their specifications to inside the parameter brackets, at the left-hand end of the list, preserving their left-to-right order, and apply the following transformation to each, according to its type:

(a) for scalar (Integer, Real(64-bit), Boolean etc) and record-type results, prefix the specification by 'VAR';

(b) for results of type STRING:

(1) decide on the maximum length of this string result which the procedure is expected to return for calls made from this module;

(2) if this has not already been done, define a type String<M>, where <M> is the maximum result length and the type is specified as PACKED ARRAY [1..<M>] OF Char (as at step 3(b))

(3) rewrite the specification (which should still be in the form 'STRING:<res_name>' as three items in the form 'VAR <res_name>: String<M>; <res_name>_max: Integer; VAR <res_name>_len: Integer'

6. (a) If the remaining result is of scalar type (Integer, Real, Boolean etc) remove the result name, to leave ': <result_type> outside the parameter list brackets. Prefix the procedure name with the keywords IMPORT FUNCTION.

(b) If the remaining result is of type RECORD, move this result specification also inside the brackets and prefix it by the keyword VAR, as in step 5(a). Prefix the procedure name with the keywords IMPORT PROCEDURE.

(c) If the remaining result is of type STRING, move it inside the brackets (at the left-hand end), and rewrite it as two items: the first (left-most) a VAR parameter of type String<M>, where <M> is the maximum length of string result expected (as for step 5 (b)(1,2)); the second an Integer whose name is formed from the result name with '_len' appended. Then add ': Integer' after the right-hand parameter list parenthesis and prefix the procedure name with the keywords IMPORT FUNCTION.

6.4 Examples

1. The process of producing an interface specification in Pascal for a Panos procedure is illustrated using the function 'GetGlobalString':

Step 1:

```
GetGlobalString (STRING: Name) INTEGER: Result, STRING: Value
```

Step 2: inapplicable

Step 3:

(a) decide on 20 (for example)
(b) define type
String20 = PACKED ARRAY[1..20] OF Char
(c) yields:

```
GetGlobalString (Name: String20; Name_len: Integer)
                INTEGER: Result, STRING: Value
```

Step 4:

```
GetGlobalString (Name: String20; Name_len: Integer)
                Result: Integer, STRING: Value
```

Step 5:

```
GetGlobalString (VAR Value: String128; Value_max: Integer;
                 VAR Value_len: Integer;
                 Name: String20; Name_len: Integer) Result: Integer
```

Step 6(a):

```
IMPORT FUNCTION GetGlobalString
                 (VAR Value: String128; Value_max: Integer;
                  VAR Value_len: Integer;
                  Name: String20; Name_len: Integer): Integer
```

An example program fragment using this procedure is:

```
VAR
  sys_ver_len, status, j: Integer;
  sys_ver: String128;
```

```
...
status := GetGlobalString
          (sys_ver, 128, sys_ver_len,
           'SYS$Version         ', 11);
IF status < 0 THEN
   WriteLn ('Couldn't get system version...');
ELSE
   BEGIN
     Write ('System version: ');
     FOR j := 1 TO sys_ver_len DO Write (sys_ver[j]);
       WriteLn
    END;
```

2. Calling a Fortran Function from Pascal (refer to the Acorn 32000 Fortran 77 Reference Manual).

An integer function FNC with two integer parameters written in Fortran might be referenced from Acorn 32000 (extended) Pascal as:

```
TYPE
   IntRef = ↑Integer;
   FNCParams = RECORD p1, p2 : IntRef END;
   FNCParamsRef = ↑FNCParams;

IMPORT FUNCTION  FNC (p: FNCParamsRef): IntRef;
```

and used as:

```
PROCEDURE LocalFNC (n1, n2: Integer): Integer;
  VAR
     rp: IntRef;  {NB rp does not point into heap, }
                  {   so no NEW/DISPOSE            }
     param: FNCParamsRef;
  BEGIN
     NEW (param);
     WITH param↑ DO BEGIN
        NEW (p1);  NEW (p2);
           p1↑ := a;  p2↑ := b;
           rp := FNC (param);
           DISPOSE (p1);  DISPOSE (p2)
     END;
     DISPOSE (param);
```

```
        LocalFNC = rp↑;   {dereference for F77 result}
    END;
```

(N.B. As described in section 6.2.2 it is necessary to compile this code with assignment checks disabled, otherwise the reference to the Fortran result variable via a Pascal pointer will probably fail since the result variable will not have been allocated on the Pascal heap.)

This is the obvious logical method of interfacing the procedure, but is by no means the most efficient way of calling a Fortran function! The main problem is that the only (legal) way of using pointers in standard Pascal is in claiming, accessing and disposing of heap memory via NEW, DISPOSE etc. - it is not possible to point pointers at normal local variables. In the above example some improvement would obtain by specifying FNC as taking a VAR parameter of type FNCParams, rather than a pointer to such a structure - VAR parameters are implemented by the same mechanism in low-level terms. Thus:

```
IMPORT FUNCTION FNC (VAR p: FNCParams): IntRef;

PROCEDURE LocalFNC (n1, n2: Integer): Integer;
   VAR
      rp: IntRef;
      paramBlock: FNCParams;
   BEGIN
      WITH paramBlock DO BEGIN
         NEW (p1);  NEW (p2);
         p1↑:= n1;  p2↑:= n2;
         rp := FNC (paramBlock);
         DISPOSE (p1);  DISPOSE (p2)
      END;
      LocalFNC = rp↑;
   END;
```

If the function is to be called more than once, it might be worthwhile removing the calls on NEW and DISPOSE from the procedure, since these involve an amount of heap management activity. This could be done by making the parameter block a global variable, and claiming the space for the parameters (via p1 and p2) only once, before the first call on LocalFNC was made. Thus we would arrive at:

```
TYPE
   IntRef = ↑Integer;
   FNCParams = RECORD p1, p2: IntRef END;
VAR
   FNCParamBlock: FNCParams;

IMPORT FUNCTION  FNC (VAR p: FNCParams): IntRef;

PROCEDURE LocalFNC (n1, n2: Integer): Integer;
   VAR
      rp: IntRef;
   BEGIN
      WITH FNCParamBlock DO BEGIN  p1↑ := a;  p2↑ := b;  END;
      rp := FNC (FNCParamBlock);
      LocalFNC = rp↑;
   END;

BEGIN
   WITH FNCParamBlock DO BEGIN NEW (p1); NEW (p2) END;
   ...
   ...
   WITH FNCParamBlock DO BEGIN DISPOSE (p1); DISPOSE (p2) END;
END.
```

In this version, the total cost of an individual call on LocalFNC has been reduced to a modest level, but at some cost in terms of program clarity. However the program does remain within the bounds of what is nominally 'legal' Pascal (ignoring the IMPORT mechanism itself). An alternative, but less 'clean' method is to use an extension of the Pascal compiler which permits the calculation of the run-time address of a variable, as below:

```
TYPE
   IntRef = ↑Integer;
   FNCParams = RECORD  p1, p2: Integer  END;

IMPORT FUNCTION FNC (VAR p: FNCParams): IntRef;

FUNCTION LocalFNC (a, b: Integer): Integer;
   VAR rp: IntRef;
       paramBlock: FNCParams;
   BEGIN
      paramBlock.p1 := ADDRESS (a);
      paramBlock.p2 := ADDRESS (b);
```

```
    rp := FNC (paramBlock);
    LocalFNC := rp↑;
END;
```

In real desperation, if speed of call were of the utmost importance, one could resort to machine code, by writing an interface routine in assembler and specifying this, rather than the actual Fortran procedure, as the external function - LocalFNC would then disappear, and the function reference might be written as:

```
IMPORT FUNCTION FNC ALIAS 'My_FNC' (a, b: Integer): Integer;
```

In assembler the interface function could be written as:

```
        MODULE   Pascal-Fortran Interface

        EXPORTC  My_FNC          ; interface procedure (defined here)

        IMPORTC  FNC             ; ref. to main function (in F77)

My_FNC
        ; on entry (via CXP) the stack looks like:
        ;
        ;   |                  |
        ;   +------------------+
        ;   |       b          | 12
        ;   +------------------+
        ;   |       a          |  8
        ;   +---------+--------+
        ;   |  x x x  |  MOD'  |  4
        ;   +---------+--------+
        ;   |        PC'       |  0      <-  SP
        ;   +------------------+

        ; create parameter vector on stack..

        ADDR    12(SP), TOS      ; address(b) as 2nd element
        ADDR    12(SP), TOS      ; address(a) as 1st

        ; pass the address of the vector as the single F77 argument

        ADDR    TOS, TOS

        ; call the function proper

        CXP     FNC
```

```
; fetch the result via the pointer returned

MOVD    0(R0), R0

; clear parameter vector space from stack

ADJSPB  =-8

; return to Pascal (or other suitable language) procedure
; - the 2 parameters to My_FNC are removed.

RXP     8

END
```

It should be noted that while the code generated in Pascal for the function LocalFNC in the preceding example may be slightly less compact than the above hand-coded version, there will be an overhead in the cost of performing the CXP/RXP through My_FNC, relative to the BSR/RET which would be performed for the call to LocalFNC. The difference between the two methods may in consequence be very small.

Appendix A

Non-Fatal Compiler error messages

```
*ERROR 1  -- this is not recognisable as a Pascal statement
```

This message is issued when the compiler cannot make any sense of the input. It may also be generated as a direct result of previous errors which cause confusion.

```
PROGRAM fail;
BEGIN
   ?Garbage
END.
```

```
*ERROR 2 -- the name of the program has been left out
```

No identifier has been found following the reserved word PROGRAM (or MODULE).

```
PROGRAM (output);
BEGIN
   writeln('no program name');
END.
```

```
*ERROR 3 -- the final item in a program must be a full stop
```

The end of a program (or module) must be marked by the reserved word 'END' followed by a full stop. Sometimes this message is issued when the compiler has got out of step with the program because of previous errors.

```
PROGRAM fail;
BEGIN
END;
```

```
*ERROR 6 -- an illegal space has been detected in <symbol>
```

The compiler has detected that an otherwise meaningless input sequence could be made understandable by removing one or more spaces. The error is usually caused by inserting spaces into the compound symbols such as < >, < =, > = etc.

```
PROGRAM fail;
VAR j,k:integer;
BEGIN
   k := 0;              {ok}
   J : = 1;             {wrong}
END.
```

```
*ERROR 7 -- digits are required here
```

This message indicates that a numerical constant has been formed incorrectly. The common cause is the omission of digits following a decimal point or following the exponential marker 'E'.

```
PROGRAM fail;
VAR r : real;
BEGIN
   r := 1.;      {should be 1.0;}
   r := 1.2e;    {should be 1.2e3 for example}
END.
```

```
*ERROR 8 -- a space is required here
```

This message is given when a reserved word immediately follows a sequence of digits without an intervening space. Even though there is no practical reason why such input must be rejected, the unfortunate choice of wording in the standard requires conforming compilers to report an error.

```
PROGRAM fail;
VAR j,k:integer;
BEGIN
   k := 19;
   IF k > 0THEN j := 12;
        {  ↑ space needed here}
   j := 123MOD k;
```

```
        {  ↑ space needed here}
END.
```

```
*ERROR 10 -- digits are required before the decimal point
```

This error is generated when the compiler detects a floating-point constant which has no leading digits. It is commonly caused by specifying constants such as one half as '.5' rather than '0.5'.

```
PROGRAM fail;
VAR r : real;
BEGIN
   r := sin(2.34e-3);
   IF r > .9 THEN writeln('strange');
END.
```

```
*ERROR 11 -- a semicolon is required here
```

The input will only make sense if the compiler assumes that a semicolon has been omitted at the indicated point.

```
PROGRAM fail(output);
VAR j : integer;
BEGIN
  j := 123
  {       ~ there should be a semicolon here}
  writeln('The square root of 123 is ', SQRT(j));
END.
```

```
*ERROR 12 -- a reference to a variable is required here
```

This message is issued when the context demands a reference to a variable and none has been found. The most common cause of this error is passing a constant when a VAR parameter was specified in the procedure heading.

```
PROGRAM fail;
PROCEDURE clear(VAR x:integer);  BEGIN
   x := 0;
END;
BEGIN
   clear(1);
END.
```

```
*ERROR 13 -- PACKED is not permitted here
```

The reserved word PACKED may only be used to qualify the type-defining words ARRAY, RECORD, SET and FILE.

```
PROGRAM fail;
TYPE thing = PACKED 1..10;       {wrong}
     pset  = PACKED SET OF char; {ok}
BEGIN
END.
```

```
*ERROR 14 -- FORWARD is not permitted here
```

The directive FORWARD must replace the body of a procedure or function in order to make the procedure or function available for use before its actual definition. FORWARD may only be used once with any given procedure or function.

```
PROGRAM fail(output);
PROCEDURE hello; FORWARD;
PROCEDURE hello; FORWARD;
PROCEDURE hello; BEGIN
   writeln('hello');
END;
BEGIN
   hello;
END.
```

```
*ERROR 15 -- <item> is required here
```

The syntax of Pascal requires that the specified symbol <item> should be found at the indicated point.

```
*ERROR 16 -- <item> should not appear in this position
```

The specified symbol <item> has been found where it was not expected.

```
PROGRAM fail;
VAR k : real;
BEGIN
   DO := k;        {DO is misplaced}
END.
```

```
*ERROR 17 -- <item> is a reserved word and may not be used as
             an identifier
```

The Pascal words which are known to the compiler: BEGIN, VAR, IF etc. are reserved and must not be used to identify objects to be used in the program

```
PROGRAM fail;
VAR from, to, high, low : integer; {TO is reserved}
BEGIN
END.
```

```
*ERROR 18 -- <item> may not be used as a set operator
```

The only permissible set operators are '+' (set union), '-' (set difference), '*' (set intersection) and the comparison operators '=', '<>', '<=' and '>='.

```
PROGRAM fail;
VAR s,t,u:SET OF char;
BEGIN
   T := ['A'..'Z'];
   U := ['a'..'z'];
   S := T / U;        {what would this mean?}
END.
```

```
*ERROR 20 -- <name> has already been declared in this block
```

This error is generated when the compiler detects that <name> is being declared for the second or subsequent time in a block. Each identifier in any block may only have a unique meaning.

```
PROGRAM fail;
VAR max : integer;      {correct declaration}
    n   : char;
    max : real;         {max cannot be both integer and real}
BEGIN
END.
```

```
*ERROR 21 -- <name> has been used in its own definition
```

This error is caused by attempting to use <name> to define itself. Note that the definition of a record may include a pointer to a record of its own type.

```
PROGRAM fail;
CONST fool = -fool;              {???}
TYPE  t = ARRAY [1..5] OF t;     {???}
BEGIN
END.
```

```
*ERROR 22 -- <name> has already been defined
```

This error is caused by attempting to define a procedure or function more than once in the same block.

```
PROGRAM fail(output);
PROCEDURE fred; BEGIN
   writeln('fred 1');
END;
PROCEDURE fred; BEGIN
   writeln('fred 2');
END;
BEGIN
   fred;
END.
```

```
*ERROR 23 -- <name> has not been declared
```

<name> has been used either before it has been declared or without any declaration.

```
PROGRAM fail;
VAR j : integer;
BEGIN
   j := k;   {k is unknown}
END.
```

```
*ERROR 24 -- <name> is not the name of a field in this record
```

<name> has been used to select a particular field from a record but the record does not contain a field called <name>.

```
PROGRAM fail;
TYPE r = RECORD  a,b,c : integer END;
VAR  x : r;
BEGIN
   x.b := 0;            {right}
   x.d := 0;            {wrong}
END.
```

```
*ERROR 25 -- <name> has already been used in this block to refer
             to an object declared in an outer block
```

This message indicates an attempt to use a name in two different ways inside the same block.

```
PROGRAM fail;
CONST max = 10;       {outer declaration}

PROCEDURE local;
CONST localmax = max;  {line A}
      max      = 20;   {line B}
BEGIN
END {of local};

BEGIN
END.
```

The procedure 'local' uses the name 'max' for two different purposes. It is the constant 10 in line A but is redefined to be the constant 20 in line B.

```
*ERROR 26 -- there is no definition for <name>
```

This message is issued when the end of a block is reached in which there was either a FORWARD declaration of <name> or a forward reference to a pointer type <name>. The error indicates that the required definition of <name> has not been found.

```
PROGRAM fail;
PROCEDURE lost; FORWARD;
BEGIN
   lost;
END.          {the definition of 'lost' is missing}
```

```
*ERROR 27 -- the bound identifier <name> may not be altered
```

The identifiers used to represent the actual bounds of a conformant array parameter are given values each time the procedure or function containing them is invoked. These values may not be altered by the program.

```
PROGRAM fail;
VAR a : ARRAY [1..5] OF char;

PROCEDURE p(a : ARRAY [low..high:integer] OF char)
BEGIN
   REPEAT
      high := high-1;   {wrong}
      a[high] := 'x';
   UNTIL high = low;
END;

BEGIN
   p(a);
END.
```

```
*ERROR 29 -- a constant is required here
```

The compiler is expecting a constant of some type.

```
PROGRAM fail;
CONST nothing = NIL;  {NIL is not valid here}
VAR a : +..19;        {+ is not a constant}
BEGIN
END.
```

```
*ERROR 30 -- an integer value is required here
```

This message is issued when a machine-code statement is incorrectly specified. Machine-code is an extension to standard Pascal.

```
*ERROR 31 -- this expression has not been formed correctly
```

This message indicates that the compiler cannot make any sense of an expression.

```
*ERROR 32 -- <name> is a procedure and may not be used in
             an expression
```

The name of a procedure is not a valid operand in an expression as a procedure call does not return a result. Procedures may only be called or passed as parameters.

```
PROGRAM fail;
VAR j : integer;

PROCEDURE twenty;
VAR result : integer;
BEGIN
   result := 20;
END;

BEGIN
   j := twenty;   {wrong}
END.
```

```
*ERROR 33 -- <item> must be followed by a boolean expression
```

This error indicates that the given <item> is only meaningful if followed by an expression which gives a boolean result but no such expression has been found.

```
PROGRAM fail(output);
VAR x : integer;
BEGIN
   x := 1;
   IF x THEN writeln('oops');   {x isn't boolean}
END.
```

```
*ERROR 34 -- the operand to the left of <item> must give a boolean value
```

The reserved words AND and OR may only follow a boolean operand. In particular they may not be used to form conjunctions or disjunctions of bit

patterns (integer values); if the extended form of the compiler is being used, the bit-vector operators '&' and '|' may be used.

```
PROGRAM fail;
VAR x : integer;
    b : boolean;
BEGIN
   IF x AND b THEN writeln('hello');   {x isn't boolean}
END.


*ERROR 35 -- the operand to the right of <item> must give
             a boolean value
```

The reserved words AND, OR and NOT must be followed by a boolean operand. In particular they may not be used to form conjunctions, disjunctions or inversions of bit patterns (integer values); if the extended form of the compiler is being used the bit-vector operators '&', '|' and '~' may be used.

```
PROGRAM fail;
VAR x : integer;
    b : boolean;
BEGIN
   IF b AND x THEN writeln('hello');
END.


*ERROR 36 -- this integer value is too large
```

This message is issued when the compiler detects that the function SQR has as its parameter an integer constant which is larger than the square root of the largest possible integer. Attempting to square this value would lead to an arithmetic fault at run-time.

```
PROGRAM fail(output);
BEGIN
   writeln(SQR(123456));   {123456 is too big for SQR}
END.
```

```
*ERROR 37 -- the value <number> is out of range
```

This message is issued when it is obvious to the compiler that PRED or SUCC must return a value which is outside the range of the parameter type. <number> will be replaced by the ordinal value of the erroneous quantity.

```
PROGRAM fail;
VAR n : (one, two, three);
BEGIN
   n := SUCC(three);   {three is too big for SUCC}
END.
```

```
*ERROR 38 -- more values are required here
```

This error is issued when a list of constants used to initialise a STATIC or EXPORT array does not contain enough values. The list must provide one value for each element of the array. STATIC and EXPORT are extensions to standard Pascal.

```
PROGRAM fail;          {needs the EXTEND option}
STATIC a : ARRAY[1..6] of integer := 1,22,333,4444,5555;
BEGIN
END.
```

```
*ERROR 39 -- a record field identifier is required here
```

A record selector (.) has been followed by something which is not an identifier.

```
PROGRAM fail;
VAR r : RECORD x,y,z : integer END;
BEGIN
   r.+ := 2;       {+ isn't a field identifier}
END.
```

```
*ERROR 40 -- a record variable is required here
```

One of the items following WITH is not a variable.

```
PROGRAM fail(output);
CONST one = 1;
BEGIN
   WITH one DO          {one is not a record variable}
      writeln('oops');
END.


*ERROR 41 -- this is not a record
```

One of the items following WITH is not a record variable.

```
PROGRAM fail(output);
VAR r : RECORD x,y,z : integer END;
BEGIN
  WITH r DO BEGIN               {ok}
     WITH y DO writeln('oops'); {y isn't a record}
  END;
END.

*ERROR 42 -- label <number> is no longer a valid destination for
             a GOTO statement
```

A GOTO statement is not permitted to transfer control into a compound statement from outside; it may only jump within a compound statement or out of it.

```
PROGRAM fail;
   LABEL 1, 2;
   VAR x : integer;
BEGIN
   IF x <> 0 THEN BEGIN
      IF x = 1 THEN GOTO 1;  {valid}
      GOTO 2;                {valid - jumping outwards}
   1: x := x+1;
   END ELSE BEGIN
      GOTO 1                 {invalid - jumping inwards}
   END;
2: x := 0;
END.
```

```
*ERROR 43 -- label <number> has already been located in this block
```

The given label has already been attached to a statement. If more than one position for a label could be specified the destination of GOTO statements would be uncertain.

```
PROGRAM fail;
LABEL 123;
VAR j,k:integer;

BEGIN
123:  j := 1;
      GOTO 123;
123:  k := 2;     {duplicate 123:}
END.


*ERROR 44 -- placing label <number> here invalidates a previous
             GOTO statement
```

A previous GOTO statement referred to the specified label which is being located on the current statement. The position of this label now means that the earlier GOTO statement is transferring control into a compound statement, which is not permitted.

```
PROGRAM fail;
   LABEL 1;
   VAR x : integer;
BEGIN
   IF x > 3 THEN GOTO 1;
   IF x = 99 THEN BEGIN
1:    x := 0;              {invalid}
   END;
END.


*ERROR 45 -- a text file is required here
```

Several of the required procedures and functions in Pascal take parameters which must be files of type TEXT. This error indicates that the given object is not a text file. Note that the type TEXT is different from a type of the form 'FILE OF char;'.

```
PROGRAM fail;
VAR f:FILE OF integer;
    x:integer;
BEGIN
   rewrite(f);              {any file will do here}
   writeln(f);              {this must be text}
END.
```

```
*ERROR 46 -- data cannot be input into an object of this type
```

A READ or GET statement has attempted to input data into an object which cannot accept values, for example a PROCEDURE or a TYPE.

```
PROGRAM fail(input);
VAR x : SET of char;
    y : RECORD z : integer END;
BEGIN
   read(x);           {wrong}
   read(y.z);         {ok}
   read(y);           {wrong}
END.
```

```
*ERROR 47 -- an item to receive input is required here
```

The procedure READ requires at least one variable to receive input data. Note that READLN may be specified with no parameters.

```
PROGRAM fail(input);
BEGIN
   readln;              {ok}
   read;                {wrong}
   readln(input);       {ok}
   read(input);         {wrong}
END.
```

```
*ERROR 48 -- a component of a packed variable may not be passed
             as a variable conformant array parameter
```

An attempt has been made to pass as a variable conformant array parameter an array which is a component of a packed structure. This is explicitly prohibited by the Standard.

```
PROGRAM fail;
TYPE tiny = 1..10;
VAR r : PACKED RECORD a : array[1..5] OF tiny END;

PROCEDURE test(VAR x : ARRAY [low..high:integer] OF tiny);
BEGIN
END;

BEGIN
   test(r.a)   {r.a is packed}
END.


*ERROR 49 -- data cannot be output from an object of this type
```

A WRITE or PUT statement has attempted to output data from an object which cannot generate values, for example a PROCEDURE or a TYPE.

```
PROGRAM fail(output);
VAR x : SET of char;
    y : RECORD z : integer END;
BEGIN
   write(x);          {wrong}
   write(y.z);        {ok}
   write(y);          {wrong}
END.


*ERROR 50 -- a field width specification is not allowed here
```

This message is issued when a parameter to write or writeln is followed by more field width specifications than are required.

```
PROGRAM fail(output);
VAR x : char;
BEGIN
   x := '?';
   writeln(x:1:2);  {only one field needed}
END.


*ERROR 51 -- an item to be output is required here
```

The procedure WRITE must be given at least one value to be output. Note that WRITELN may be specified without any parameters.

```
PROGRAM fail(output);
BEGIN
   write;           {wrong}
   writeln;         {ok}
   write(output);   {wrong}
   writeln(output); {ok}
END.
```

```
*ERROR 52 -- a file variable is required here
```

When the procedures WRITE, WRITELN, READ, READLN, RESET, REWRITE and PAGE, or the functions EOLN and EOF are used, they will either use a default file (input or output), or the file must be specified as the first or only parameter.

```
PROGRAM fail;
VAR j : integer;
BEGIN
   rewrite(j);   {j isn't a file}
END.
```

```
*ERROR 53 -- field widths must be greater than zero
```

When writing to text files, a field width may be specified as an integer expression following a colon. All such expressions must return values which are strictly greater than zero. This message indicates that a field width is less than one. Note that this error can only be generated if the field width expression is a constant; when a more complex expression is given, the error has to be detected at run-time.

```
PROGRAM fail(output);
BEGIN
   writeln('oops':0);
END.
```

```
*ERROR 54 -- an expression for an element of the set is required here
```

The set constructor [...] must contain a list of expressions, each of which identifies an item in the set. The error is usually caused by a spurious comma within the set constructor.

```
PROGRAM fail;
VAR S : SET OF 1..9;
BEGIN
   s := [1,2,,3];
END.
```

```
*ERROR 55 -- the ordinal values of components of sets are restricted
       to being in the range 0..255
```

The Pascal standard does not prescribe any limits on the range of ordinal values that components of sets may take. This implementation restricts the ordinal values of all components of sets to the range 0..255.

```
PROGRAM fail;
TYPE big   = 1..1000;
     small = 5..30;
VAR bs : SET OF big;     {too big}
    ss : SET OF small;   {ok}
BEGIN
END.
```

```
*ERROR 56 -- the name of a type is required here
```

This error indicates that the compiler expected an identifier which has been, or is to be declared as a type. The common cause of such errors is attempting to define types explicitly (e.g. 1..10) rather than giving them a name in a TYPE declaration first.

```
PROGRAM fail;
TYPE small = 1..10;
     7 = 1..7;              {7 is not an identifier}
VAR  p :↑1..12;             {↑ needs a type identifier}
     r : RECORD
           CASE x :1..2 OF {CASE needs a type identifier}
             1: (a : real);
             2: (b : integer)
         END;
BEGIN
END.
```

```
*ERROR 57 -- the objects in the set constructor do not all have
             the same type
```

When square brackets are used to construct a set value, the expressions contained within must all yield values of the same type.

```
PROGRAM fail;
TYPE t1 = (this, that, other);
     t2 = (here, there, everywhere);
VAR s : SET OF t1;
BEGIN
   s := [this, there, that];  {'there' is the wrong type}
END.
```

```
*ERROR 58 -- the operands of <item> are of unsuitable types
```

The operator <item> has been given operands to which it cannot be applied.

```
PROGRAM fail;
VAR a, b, c : char;
BEGIN
   a := b + c;  {chars cannot be added together}
END.
```

```
*ERROR 59 -- the operands for <item> are of incompatible types
```

The operator <item> has been used to operate upon operands of different and incompatible types.

```
PROGRAM fail;
VAR c : boolean;
    s : SET OF char;
BEGIN
   IF c IN s THEN writeln;  {c is not a char}
END.
```

```
*ERROR 60 -- this type does not define a range of values
```

Certain types in Pascal, known as 'ordinal types', define ranges of values. These types are: integer, char, boolean and all enumerated types. In

addition, any type which is a subrange of these is an ordinal type. Whenever a type is used to select an object from a number of objects, an ordinal type is required.

```
PROGRAM ok;
   TYPE colour = (red, yellow, green, blue);
   VAR vector : ARRAY [colour] OF integer;
BEGIN
END.
```

The error message is generated if an enumerated type is required and the given type is not enumerated.

```
PROGRAM fail;
   TYPE enum1 = (alpha, beta, gamma);
   VAR  bad1 : ARRAY [real] OF integer;  {real isn't ordinal}
        good : SET OF enum1;             {enum1 is ordinal}
BEGIN
END.


*ERROR 61 -- integer index types are not permitted
```

The compiler will not permit arrays to be declared where the type of the index is integer. If it were to do so the program would require more working memory than most computers can provide.

```
PROGRAM fail;
VAR a : ARRAY [integer] of char;  {this would be huge}
BEGIN
   a[-maxint] := 'a';
   a[+maxint] := 'o';
END.


*ERROR 62 -- an object of this type may not be given a '+' sign

*ERROR 62 -- an object of this type may not be given a '-' sign
```

Plus or minus signs may only be given to numerical objects. In particular enumerated type and char type values may not be signed.

```
PROGRAM fail;
TYPE  ok   = (red, yellow, green, blue);

PROCEDURE thing;
CONST bad1 = -'a';
      bad2 = +green;
BEGIN
END;

BEGIN
END.


*ERROR 63 -- this type must not have a component which is a file
```

In certain contexts Pascal places the restriction on the type of acceptable objects that they must not have any component which has the type FILE.

```
PROGRAM fail;
TYPE rf = RECORD j : integer;
                 f : FILE OF real
          END;
VAR a, b : rf;
BEGIN
   a := b;  {cannot copy records containing files}
END.


*ERROR 64 -- a type specification is required here
```

This message indicates that a general type specification is needed, that is, either the name of a type or an explicit type definition such as 1..10.

```
PROGRAM fail;
VAR j,k:;           {no type given}
BEGIN
END.


*ERROR 65 -- the identifier of a type which defines a range of values is
       required here
```

Certain types in Pascal, known as 'ordinal types', define ranges of values. These types are: integer, char, boolean and all enumerated types. In addition any type which is a subrange of these is an ordinal type. Whenever

a type is used to select an object from a number of objects an ordinal type is required. This message is issued when the identifier of such a type is needed but has not been given.

```
PROGRAM fail;
VAR a : ARRAY [2..8] OF integer;

PROCEDURE p(w : ARRAY [min..max:real] OF integer);
   {real is not an ordinal type !}
BEGIN
END;

BEGIN
   p(a);
END.


*ERROR 66 -- the type of result <name> returns must be defined here
```

<name> is being defined as a function which means that when it is invoked it will return a value as its result. The type of that result must be specified at the indicated point.

```
PROGRAM fail;

FUNCTION unknown(x:integer); BEGIN
         {result type here !}
   unknown := x-1;
END;

BEGIN
END.


*ERROR 67 -- the type of <name> was given in a previous FORWARD
             declaration and must not be repeated here
```

<name> has already been declared as a function in a FORWARD declaration at which time the type of result it returns must have been defined. The current statement is the actual definition of the function. Sadly Pascal forbids the repetition of the result type under these circumstances.

```
PROGRAM fail(output);

FUNCTION two:integer; FORWARD;

PROCEDURE show; BEGIN
   writeln(two);
END;

FUNCTION two:integer;          {not permitted}
BEGIN
   two := 22222;
END;

BEGIN
   show;
END.
```

```
*ERROR 68 -- the type of result returned by <name> does not
             match a previous FORWARD declaration
```

This message is issued when the type of a function is different from a previous specification of that function. The error is an additional indication of trouble as it is only ever generated under the circumstances when error 70 will also be issued.

```
PROGRAM fail(output);

FUNCTION two:integer; FORWARD;

PROCEDURE show; BEGIN
   writeln(two);
END;

FUNCTION two:real;          {not permitted}
                            {and the wrong type}
BEGIN
   two := 22.222;
END;

BEGIN
   show;
END.
```

```
*ERROR 69 -- this is not a pointer type
```

This message is issued when NEW or DISPOSE are called with parameters which are not pointers.

```
PROGRAM fail;
VAR x:integer;
BEGIN
   NEW(x);        {x isn't a pointer}
END.
```

```
*ERROR 70 -- the type of the expression following := is unsuitable
             for assigning to the variable to the left of :=
```

An assignment statement has been found which attempts to assign a value to a variable whose type is incompatible with the type of the value.

```
PROGRAM fail;
VAR j : integer;
    r : real;
    s : SET OF char;
BEGIN
   j := r;    {wrong}
   r := j;    {right}
   s := 'c';  {wrong - 'c' is not a set}
END.
```

```
*ERROR 71 -- the type of result <name> returns may be only
             a simple type or a pointer type
```

Functions may only return results which are of a simple type (integer, real, char, boolean, enumerated or a subrange) or a pointer type.

```
PROGRAM fail;
TYPE stype = SET OF char;
VAR s : stype;

FUNCTION f:stype;        {wrong - sets aren't simple}
BEGIN
   f := [];
END;
```

```
BEGIN
   s := f;
END.


*ERROR 72 -- the upper and lower limits of this subrange are of different
       types
```

When defining a subrange type the upper and lower limits of the range must be constants of the same type.

```
PROGRAM fail;
TYPE day = (sun, mon, tue, wed, thurs, fri, sat);
     season = (spring, summer, autumn, winter);
     range  = mon..winter;            {very strange}
BEGIN
END.


*ERROR 73 -- the upper limit of the subrange is less than the
             lower limit
```

This message indicates that the limits of a subrange definition are inside-out, that is, the lower limit is greater than the upper limit. This would imply that the subrange contains no members.

```
PROGRAM fail;
TYPE backwards = 10..1;
BEGIN
END.


*ERROR 74 -- the upper limit of the subrange is required here
```

This error indicates that the upper limit of a subrange definition has either been omitted or specified incorrectly.

```
PROGRAM fail;
TYPE bad = 1..;
BEGIN
END.
```

```
*ERROR 75 -- sets may not be compared in this way
```

The comparison operators '<' and '>' may not be used to compare set values.

```
PROGRAM fail(output);
VAR s1, s2 : SET OF boolean;
BEGIN
   IF s1 <= s2 THEN writeln('right');
   IF s1 <  s2 THEN writeln('wrong');
END.
```

```
*ERROR 76 -- sets may only be compared to sets
```

This message is caused by an attempt to compare a set value with another value which is not a set.

```
PROGRAM fail(output);
VAR s : SET OF 1..10;
    t : 1..10;
BEGIN
   IF s = t THEN writeln('wrong');
END.
```

```
*ERROR 77 -- incompatible sets may not be compared
```

Two set values may only be compared if they are made of items of the same type.

```
PROGRAM fail(output);
VAR s1 : SET OF char;
    s2 : SET OF boolean;
BEGIN
   IF s1 = s2 THEN writeln('wrong');
END.
```

```
*ERROR 78 -- values of incompatible types may not be compared
```

Two values may only be compared if their types are compatible.

```
PROGRAM fail(output);
VAR j : real;
    c : char;
BEGIN
   IF j <> c THEN writeln('wrong');
END.
```

```
*ERROR 79 -- <item> may not be compared
```

The standard does not permit the comparison of file or record variables.

```
PROGRAM fail(output);
VAR f1, f2 : FILE OF integer;
    r1, r2 : RECORD x,y,z : integer END;
BEGIN
   IF f1 = f2 THEN writeln('wrong');
   IF r1 = r2 THEN writeln('wrong');
END.
```

```
*ERROR 80 -- <item> may not be used to compare pointers
```

Pointer types may only be tested for equality or inequality.

```
PROGRAM fail(output);
VAR p1, p2 : ↑integer;
BEGIN
   IF p1  = p2 THEN writeln('equal');       {ok}
   IF p1 <> p2 THEN writeln('different');   {ok}
   IF p1 <= p2 THEN writeln('strange');     {wrong}
END.
```

```
*ERROR 81 -- no statement has been marked with label <number>
```

This message is issued at the end of a block which contained the declaration of the label <number> but did not use that label to identify a statement.

```
PROGRAM fail(output);
   PROCEDURE test;
      LABEL 10, 20;
      BEGIN
   10: writeln('label 10 identifies this statement');
      END;  {no statement labelled 20: }
BEGIN
END.


*ERROR 82 -- labels must be in the range 0..9999
```

The Pascal standard states that labels are distinguished by their apparent numerical value and that this value must be in the range 0..9999.

```
PROGRAM fail(output);
LABEL 1,                    {ok}
      10000;                {wrong}
BEGIN
1:     writeln('ok');
10000: writeln('bad');
END.


*ERROR 83 -- a label is required here
```

In Pascal a label must be an integer constant in the range 0..9999. This message is issued when such a label is required but has not been found.

```
PROGRAM fail(output);
LABEL x;              {x is not an integer constant}
BEGIN
   GOTO x;            {ditto}
x: writeln('here');   {and ditto}
END.


*ERROR 84 -- this case label has already been used in this CASE
             statement
```

This message is issued when a constant is used to label a statement for a second or subsequent time.

```
PROGRAM fail(input, output);
TYPE t = 1..3;
VAR x : t;
BEGIN
   read(x);
   CASE x of
1:   writeln('one');
2:   writeln('two');
3:   writeln('three');
2:   writeln('oops!');   {duplicate 2:}
   END;
END.


*ERROR 85 -- this case label is out of range
```

Surprisingly, the Pascal standard does not proscribe the specification of case labels which are outside the range of the case selector except when the CASE is used to define record variants.

```
PROGRAM fail(output,input);
TYPE small = 1..3;
     rec   = RECORD CASE s:small OF
                        1: (a:integer);
                        2: (b:char);
                        3: (c:real);
                        4: (d:boolean); {error 4 is not in small}
             END;
VAR tiny:small;
BEGIN
   read(tiny);
   CASE tiny OF
      1: writeln('one');
      2: writeln('two');
      3: writeln('three');
      4: writeln('four');  {warning 4 is not in small}
   END;
END.
```

```
*ERROR 86 -- a case constant is required here
```

This message indicates that the compiler expected a case label to be appended to the current statement. This is commonly caused by forgetting to bracket several statements with BEGIN and END when all of those statements are to be executed for the selected case.

```
PROGRAM fail(input, output);
VAR tiny:1..3;
BEGIN
   read(tiny);
   CASE tiny OF
      1: write('one');
         writeln('and only');  {no label?}
      2: writeln('two');
      3: writeln('three');
   END;
END.
```

```
*ERROR 87 -- the type of the case constant differs from the type
             of the case index
```

This message is issued when the value used to identify one of a list of statements in a CASE construction is not of the same type as the value used to select the particular case.

```
PROGRAM fail(output);

TYPE month = (jan, feb, mar, apr, may);
     season = (spring, summer, autumn, winter);
VAR date : month;

BEGIN
  CASE date of                {date is a 'month'-type thing}
     winter: writeln('cold'); {winter is a 'season'-type thing}
     summer: writeln('warm')  {so is summer}
  END;
END.
```

```
*ERROR 88 -- the item used to select a particular case
            is not of a type which defines a range of values
```

Certain types in Pascal, known as 'ordinal types', define ranges of values. These types are: integer, char, boolean, enumerated types and all subranges. Whenever a type is used to select an object from a number of objects an ordinal type is required, in particular the value used as a selector in a CASE statement must be of ordinal type.

```
PROGRAM fail(output);
VAR r : real;    {this isn't ordinal}
BEGIN
   CASE r OF
1:  writeln('one');
3:  writeln('three');
   END;
END.
```

```
*ERROR  89 -- the case statement did not include all possible cases
```

A case statement has been found which has not provided a meaning for all the possible values of the case selector. When this occurs in a RECORD definition it is an error. When it occurs in a CASE statement it is only a warning, however if the case selector takes one of the unspecified values during execution a run-time error will be reported.

```
PROGRAM fail(output,input);
TYPE small = 1..3;
     rec   = RECORD CASE s:small OF
                      1: (a:integer);
                      3: (b:real);
             END;          {error - no case 2}
VAR tiny:small;
BEGIN
   read(tiny);
   CASE tiny OF
      1: writeln('one');
      3: writeln('three');
   END;                          {warning - no case 2}
END.
```

```
*ERROR 91 -- this is not a valid case selector
```

This message is issued when the extended forms of NEW or DISPOSE are used and the extra parameters do not correspond to the variables in the CASE parts of the record description.

```
PROGRAM fail;
   TYPE three = 1..3;
        rec = RECORD CASE x:three of
                         1:(a : integer);
                         2:(b : real);
                         3:(c : char);
              END;
       point = ↑rec;
   VAR thing : point;
BEGIN
   new(thing, 2);            {2 is ok for x above}
   new(thing, 4);            {but 4 is not}
END.
```

```
*ERROR 92 -- this statement threatens to alter the loop control
             variable <name>
```

The control variable of a FOR loop may not be used inside the body of the loop in such a way as to threaten to alter it. This means that the variable cannot appear to the left of :=, as a VAR parameter or as the control variable of a nested FOR statement. Note that the error only indicates that the variable is threatened; it is still an error even though the variable can never actually be altered.

```
PROGRAM fail(input);
VAR j, k : integer;
BEGIN
   FOR j := 1 to 10 DO BEGIN
      read(k)
      IF k = 0 THEN j := 10   {wrong}
   END;
END.
```

```
*ERROR 93 -- the name of the control variable is required here
```

The name of the variable to be used to control a FOR loop must be specified immediately following the word FOR.

```
PROGRAM fail(output);
BEGIN
   FOR 1 TO 10 DO          {no control variable}
      writeln('hello');
END.
```

```
*ERROR 94 -- this is not a variable and so may not be used to
             control the FOR statement
```

A FOR statement must use a variable declared in the current block to control the repeated execution of the controlled statement. This error indicates that an attempt has been made to use something other than a variable as the control variable.

```
PROGRAM fail(output);
TYPE cvar = 1..10;
BEGIN
   FOR cvar := 1 TO 10 DO   {cvar is not a variable}
      writeln('hello');
END.
```

```
*ERROR 95 -- the control variable <name> was not declared in
             this block
```

The FOR loop control variable must be a local variable, that is, it must have been declared in the VAR section of the current block. Note in particular that a formal parameter to a procedure or function may not be used as a control variable.

```
PROGRAM fail(output);
VAR global : integer;

PROCEDURE print; BEGIN
  FOR global := 1 to 10 DO   {global not declared in 'print'}
      writeln('printing');
END;
```

```
BEGIN
   print;
END.


*ERROR 96 -- the control variable <name> is not of a type
             which defines a range of values
```

The control variable of a FOR statement must have a type which defines a range of values (an ordinal type). This type must be integer, char, boolean, enumerated type, or a subrange type.

```
PROGRAM fail(output);
VAR x : real;
BEGIN
   FOR x := 1 TO 10 DO  {x isn't ordinal}
      writeln('hello');
END.


*ERROR 97 -- either TO or DOWNTO is required here
```

When specifying a FOR statement the initial and final values for the control variable must be separated by either TO or DOWNTO.

```
PROGRAM fail(output);
VAR j:integer;
BEGIN
   FOR j := 1, 10 DO
       writeln('hello'):
END.


*ERROR 98 -- the initial value cannot be assigned to <name>
```

The initial value of a FOR loop cannot be assigned to the loop control variable <name> because it is not of a suitable type or not in the correct range.

```
PROGRAM fail(output);
TYPE days =0(sun, mon, tue, wed, thurs, fri, sat);
     week = mon..fri;
VAR w : week;
BEGIN
```

```
   FOR w := sun TO fri DO writeln('day');  {not in range}
   FOR w :=  1  TO fri DO writeln('day');  {wrong type}
END.
```

```
*ERROR 99 -- the final value cannot be assigned to <name>
```

The final value of a FOR loop cannot be assigned to the loop control variable <name> because it is not of a suitable type or not in the correct range.

```
PROGRAM fail(output);
TYPE days = (sun, mon, tue, wed, thurs, fri, sat);
     week = mon..fri;
VAR w : week;
BEGIN
   FOR w := mon TO sat DO writeln('day');  {not in range}
   FOR w := mon TO  7  DO writeln('day');  {wrong type}
END.
```

```
*ERROR 100 -- a procedure or function contains a statement which
              threatens to alter the loop control variable <name>
```

Note that this error is commuted into a warning if the EXTEND option is specified.

The variable which is used to control the execution of a FOR statement may not be used inside a procedure or function in such a way as to threaten to alter it, even if the variable will not in fact be altered during the execution of the loop.

```
PROGRAM fail(output);
VAR x : integer;
PROCEDURE proc;
BEGIN
   X := 0          {this is the threatening statement}
END;
BEGIN
   FOR x := 1 to 10 DO writeln('hello');     {incorrect}
END.
```

```
*ERROR 101 -- parameters must be defined here
```

The declaration of a formal parameter is expected here. The common cause for this message is a spurious semicolon separating formal parameters.

```
PROGRAM fail;
PROCEDURE thing(x : integer; ; z : real);
BEGIN          {nothing here !}
END;
BEGIN
END.
```

```
*ERROR 102 -- the parameters required by <name> were defined in a previous
              FORWARD declaration and must not be repeated here
```

When a procedure or function is declared as being FORWARD the parameters it requires must be specified at that point. These parameters must not be specified again when the actual definition of the procedure or function is made.

```
PROGRAM fail;
PROCEDURE thing(x : integer); FORWARD;
PROCEDURE thing(x : integer);          {credible, but wrong}
BEGIN
END;
BEGIN
END.
```

```
*ERROR 103 -- a parameter in brackets is required here
```

A procedure or function has been invoked with no parameters when its definition required at least one.

```
PROGRAM fail;
VAR x : integer;
BEGIN
   x := ABS;           {ABS needs one parameter}
END.
```

```
*ERROR 104 -- this statement requires that the name INPUT appear in
       the list of parameters following PROGRAM
```

If the file parameter is omitted from a call on the procedures READ or READLN, or the functions EOLN or EOF, the required file INPUT is assumed. In such cases INPUT must appear as one of the program parameters at the head of the program. However it must not appear in an outermost VAR statement.

```
PROGRAM fail;
BEGIN
   readln;   {implies use of INPUT}
END.
```

```
*ERROR 105 -- this statement requires that the name OUTPUT appear
       in the list of parameters following PROGRAM
```

If the file parameter is omitted from calls on the procedures WRITE, WRITELN or PAGE the required file OUTPUT is assumed. In such cases OUTPUT must appear as one of the program parameters at the head of the program. However it must not appear in an outermost VAR statement.

```
PROGRAM fail;
BEGIN
  writeln('hello'); {implies use of OUTPUT}
END.
```

```
*ERROR 106 -- the field width parameters must be of integer type
```

The parameters used in WRITE and WRITELN to specify field widths must yield integer values.

```
PROGRAM fail(output);
BEGIN
   writeln('hello':9.5);
END.
```

```
*ERROR 107 -- ABS needs a real or integer type parameter
```

The function ABS returns the absolute value of either integer or real values; it cannot be used with any other types.

```
PROGRAM fail;
VAR c:char;
BEGIN
   c := ABS('x');
END.
```

```
*ERROR 108 -- SQR needs a real or integer type parameter
```

The function SQR returns the square of either integer or real values; no other type of value may be given as the parameter.

```
PROGRAM fail(output);
BEGIN
   writeln(sqr(FALSE));
END.
```

```
*ERROR 109 -- CHR needs an integer type parameter
```

The parameter given to CHR may only be of type integer or a subrange of integer.

```
PROGRAM fail;
VAR c : char;
    x : real;
BEGIN
   x := 1;
   c := CHR(x);   {x isn't an integer}
END.
```

```
*ERROR 110 -- ODD needs an integer type parameter
```

The function ODD may only be given a parameter of type integer or a subrange of integer.

```
PROGRAM fail(output);
VAR b : boolean;
BEGIN
   IF ODD(b) THEN writeln('not even');
END.
```

```
*ERROR 111 -- PRED needs a parameter of a type which defines
              a range of values
```

The parameter given to PRED may only be of an ordinal type, that is integer, char, boolean, an enumerated type or a subrange.

```
PROGRAM fail;
VAR val : real;
BEGIN
   val := PRED(val);
END.
```

```
*ERROR 112 -- SUCC needs a parameter of a type which defines
              a range of values
```

The parameter given to SUCC may only be of an ordinal type, that is integer, char, boolean, an enumerated type or a subrange.

```
PROGRAM fail;
VAR val : real;
BEGIN
   val := SUCC(val);
END.
```

```
*ERROR 113 -- TRUNC needs a real type parameter
```

The parameter given to TRUNC must be of real type. Note that even though an integer value can often be used in the context of a real value the standard does not permit integer parameters for TRUNC.

```
PROGRAM fail;
VAR x    : real;
    n, m : integer;
BEGIN
   x := 1.54;
   m := TRUNC(x);
   n := TRUNC(m);              {sadly, not permitted}
   m := TRUNC(TRUE);           {just rubbish}
END.
```

```
*ERROR 114 -- ROUND needs a real type parameter
```

The parameter given to ROUND must be of real type. Note that even though an integer value can often be used in the context of a real value the standard does not permit integer parameters for ROUND.

```
PROGRAM fail;
VAR x    : real;
    n, m : integer;
BEGIN
   x := 1.54;
   m := ROUND(x);
   n := ROUND(m);              {sadly, not permitted}
   m := ROUND(TRUE);           {just rubbish}
END.
```

```
*ERROR 115 -- a real or integer type parameter is needed here
```

This error is issued when a required function is called with a parameter which does not give either an integer or a real value.

```
PROGRAM fail;
VAR x : real;
BEGIN
   x := SIN('s');  {'s' is neither integer nor real}
END.
```

```
*ERROR 116 -- this parameter does not match the specification
              given in the declaration of <name>
```

This message is issued when an actual parameter given to a procedure or function, <name>, is not of the correct type as defined by the declaration of <name>.

```
PROGRAM fail;
VAR x : real;

PROCEDURE thing(VAR w : integer);
BEGIN
END;

BEGIN
```

```
   thing(x);  {x is not an integer variable}
END.
```

```
*ERROR 117 -- the parameters required by <name> are different from
        those specified in the definition of this parameter
```

This message is issued when an attempt is made to pass a procedure or function as a parameter to another procedure or function, <name>, and the parameters required by the actual parameter do not match those required by the formal parameters of the parameter of <name>. Note that the standard is very strict about the matching.

```
PROGRAM fail(output);

PROCEDURE actual1(x : integer; y : integer); BEGIN
   writeln('actual1');
END;

PROCEDURE actual2(x, y : integer); BEGIN
  {NOTE - these parameters are DIFFERENT from actual1's}
   writeln('actual2');
END;

PROCEDURE actual3(w : real); BEGIN
   writeln('actual3');
END;

PROCEDURE try(a : integer;
              PROCEDURE formal(p, q : integer);
              c : integer);
BEGIN
   formal(a, c);
END;

BEGIN
   try(1, actual1, 3); {strictly wrong}
   try(1, actual2, 3); {right}
   try(1, actual3, 3); {wrong}
END.
```

```
*ERROR 118 -- <name> does not require any parameters
```

<name> is the identifier of a procedure or function which has been specified with a list of parameters even though the definition of <name> did not include any parameters.

```
PROGRAM fail(output);

PROCEDURE thing; BEGIN
  writeln('this is thing');
END;

BEGIN
   thing(output);  {thing doesn't take parameters}
END.
```

```
*ERROR 119 -- more parameters have been given than are required
              by <name>
```

The parameters given to a procedure or function reference must match the formal parameters in the definition in number, position and type. This reference to <name> has been given too many parameters.

```
PROGRAM fail(output);

PROCEDURE show(x, y : integer); BEGIN
   writeln('x+y =', x+y);
END;

BEGIN
   show(1, 2, 3);   {only two parameters needed}
END.
```

```
*ERROR 120 -- <name> needs a list of parameters in brackets here
```

The procedure of function <name> has been specified without any parameters when the definition of <name> included at least one parameter.

```
PROGRAM fail(output);

PROCEDURE show(x : integer); BEGIN
   writeln('x=', x);
END;
```

```
BEGIN
   show;              {parameter needed here}
END.


*ERROR 121 -- program parameter <name> has not appeared in a
              VAR statement
```

<name> was specified in the parameter list following the initial PROGRAM statement. Except in the case of INPUT and OUTPUT all such parameters must also appear in the outermost VAR statement so that they may be given a type.

```
PROGRAM fail(output, data);

{should be VAR data:text here}

BEGIN
   readln(data);       {data has not been defined}
END.


*ERROR 122 -- program parameter <name> is restricted to being
              a file
```

This implementation restricts the type of the objects which appear in the parameters following the initial PROGRAM statement to being files. An attempt to give them any other type will invoke this message.

```
PROGRAM fail(thing);
VAR thing : integer;
BEGIN
END.


*ERROR 123 -- program parameter <name> may not be a file of
              pointers
```

This implementation of Pascal prohibits the use of files of pointers as program parameters.

```
PROGRAM fail(ipoint);
TYPE pi = ↑integer;
VAR ipoint : FILE OF pi;
BEGIN
END.
```

```
*ERROR 124 -- this parameter must have packed type
```

The first parameter of UNPACK and the third parameter of PACK must be references to arrays which have been declared PACKED.

```
PROGRAM fail;
VAR p :  ARRAY [1..5] OF char;
    u :  ARRAY [1..5] OF char;
BEGIN
   pack(u, 1, p);   {p is not PACKED}
   unpack(p, u, 1); {p is not PACKED}
END.
```

```
*ERROR 125 -- this parameter must not have packed type
```

The first parameter of PACK and the third parameter of UNPACK must not be references to arrays which have been declared PACKED.

```
PROGRAM fail;
VAR p : PACKED ARRAY [1..5] OF char;
    u : PACKED ARRAY [1..5] OF char;
BEGIN
   pack(u, 1, p);     {u is PACKED}
   unpack(p, u, 1);   {u is PACKED}
END.
```

```
*ERROR 126 -- this item may not be passed as a VAR parameter
```

The variable used to define the currently active variant of a record may not be passed to a procedure or function as a VAR parameter.

```
PROGRAM fail;
VAR r : RECORD
           CASE x : boolean OF
            true: (t : integer);
            false:(f : real);
        END;
PROCEDURE test(VAR v : boolean);
BEGIN
END;

BEGIN
```

```
  test(r.x);        {r.x is a case selector}
END.
```

```
*ERROR 127 -- PROGRAMs may not export procedures or functions
```

A compilation unit that starts with PROGRAM is not permitted to define EXPORT PROCEDUREs or EXPORT FUNCTIONs, however, it may export variables. This is an extension to standard Pascal.

```
PROGRAM fail;

EXPORT PROCEDURE thing;
BEGIN
END;

BEGIN
   thing;
END.
```

```
*ERROR 128 -- a procedure identifier is required here
```

In the definition of a procedure the name of the procedure must immediately follow the word PROCEDURE.

```
PROGRAM fail;
PROCEDURE; BEGIN
END;
BEGIN
END.
```

```
*ERROR 129 -- a function identifier is required here
```

In the definition of a function the name of the function must immediately follow the word FUNCTION.

```
PROGRAM fail;
FUNCTION:integer; BEGIN
END;
BEGIN
END.
```

```
*ERROR 130 -- declarations must appear in the order
       LABEL, CONST, TYPE, VAR, PROCEDUREs and FUNCTIONs
```

Pascal defines that the order of declarations must be: LABELs then CONSTs then TYPEs then VARs then PROCEDUREs and FUNCTIONs in any order.

```
PROGRAM fail;
TYPE t = 1..10;
VAR a : t;
VAR b : integer; {wrong - repeated VAR}
CONST one = 1;   {wrong - just out of order}
BEGIN
END.
```

```
*ERROR 131 -- a variable, function or procedure identifier is
              required here
```

The compiler is expecting a statement of the form:

```
variable := expression;
procedureid;
```

or

```
functionid := expression;
```

```
PROGRAM fail;
TYPE thing = 1..10;
Var x : thing;
BEGIN
   x := thing;
END.
```

```
*ERROR 132 -- <name> may not be passed as a procedural parameter
```

The Pascal standard does not allow any of the required procedures (or functions) to be passed as procedural (or functional) parameters. Function results, however, may be passed as value parameters.

```
PROGRAM fail(output);

PROCEDURE try(FUNCTION x(n : integer):integer); BEGIN
   writeln(x(123));
END;

BEGIN
   try(PRED);   {wrong}
   try(SQR);    {wrong}
END.


*ERROR 133 -- this function cannot execute an assignment
       of the form: <name> := result
```

All functions must contain at least one executable statement of the form:

```
   functionid := expression;
```

in order that the result of the function shall be defined. The error message indicates that either no such statement occurred at all, or that it is known to the compiler that none can ever be executed.

```
PROGRAM fail;
CONST flag = 1;
VAR   x : integer;

FUNCTION one:integer
BEGIN
   IF flag = 0 THEN one := 1;  {can never be executed}
END;

BEGIN
   x := one;
END.


*ERROR 134 -- the assignment of a value to <name> may occur only
       within the definition of the function <name>
```

The result of a function is assigned by a special form of the assignment statement in which the name of the function appears on the left-hand side, This form of assignment is only permitted within the definition of the function, and will be faulted if it occurs in any other context.

```
PROGRAM fail;
   VAR n : integer;
   FUNCTION one:integer;
   BEGIN
      one := 1;          {correct}
   END;
BEGIN
   n := one;             {correct}
   one := 1;             {incorrect - not inside 'one'}
END.
```

```
*ERROR 135 -- function <name> cannot be used in this way
```

This message is issued when a required function is used as though it were a variable.

```
PROGRAM fail;
BEGIN
   sqrt := 0;
END.
```

```
*ERROR 136 -- the type of the array index does not match the type
              given in the array declaration
```

The expression used to select an element of an array must give a value which is of the same type as was used to declare the array.

```
PROGRAM fail(output);
   TYPE person = (fred, jill, anne, jim);
   VAR who : person;
       lucky : ARRAY [person] of boolean;
BEGIN
   IF lucky[who]  THEN writeln('yes')
                  {OK - who is 'person'-type}
   IF lucky[true] THEN writeln('no')
                  {wrong - true is not 'person'-type}
END.
```

```
*ERROR 137 -- the array has been given more subscripts than are
              required by its declaration
```

The declaration of every array defines the number of expressions that are required to select an element from the array. This error indicates that too many subscripts have been given. Note that giving a smaller number of subscripts than the maximum permitted by the declaration is valid; it will result in an object which is itself an array.

```
PROGRAM fail;
VAR s : ARRAY [1..3] of PACKED ARRAY [1..4] OF char;
BEGIN
   s[1]     := 'abcd';    {ok}
   s[1][2]  := '?';       {ok}
   s[1,2]   := '?';       {ok - same as the previous assignment}
   s[1,2,3] := '.';       {wrong}
END.
```

```
*ERROR 138 -- an array type is required here
```

The first and third parameters of PACK, and the first and second parameters of UNPACK must be of type ARRAY.

```
PROGRAM fail;
VAR a : ARRAY [1..5] of char;
    x : char;
BEGIN
   pack(a, 1, x);          {x isn't an array}
   unpack(x, a, 1);        {ditto}
END.
```

```
*ERROR 139 -- the type of this parameter is not the same as the
       type of the previous parameter but the declaration
       of these parameters requires them to have identical types
```

When a conformant array parameter is declared two names are reserved to hold the lower and upper bounds of the actual array passed. If the declaration specifies that two or more parameters share these names than the arrays actually passed in these positions must have the same bounds.

```
PROGRAM fail;
   VAR a, b : ARRAY [1..5] of integer;
   VAR c    : ARRAY [2..6] of integer;

PROCEDURE unique( x : ARRAY[low..high : integer]
                          of integer;
                  y : ARRAY[small..big : integer]
                          of integer)
BEGIN
END;

PROCEDURE share(x, y : ARRAY[low..high : integer]
                           of integer)
BEGIN
END;

BEGIN
   unique(b, a);   {ok}
   unique(a, c);   {ok}
   unique(c, c);   {ok}
    share(b, a);   {OK  a & b have the same bounds}
    share(a, c);   {not OK  a & c have different bounds}
    share(c, c);   {OK - they must be the same}
END.


*ERROR 140 -- more values have been supplied than are required by
              the array
```

This error is issued when the list of constants used to initialise an EXPORT or STATIC array contains more values than the array has elements. EXPORT and STATIC are extensions to standard Pascal.

```
PROGRAM fail;
STATIC a : ARRAY [1..3] OF char := 'a', 'b', 'c', 'd';
BEGIN
END.


*ERROR 141 -- PACK and UNPACK may only be used to move data between
       arrays whose elements are of identical type
```

This message is issued when an attempt is made to use PACK or UNPACK to assign values of an unsuitable type to the destination array.

```
PROGRAM fail;
VAR p : PACKED ARRAY [1..5] OF char;
    u :        ARRAY [2..6] OF boolean;
BEGIN
   pack(u, 2, p);  {boolean can't be assigned to char}
END.


*ERROR 142-- the type of this expression is not suitable as an
             index into the unpacked array
```

This error indicates an attempt to use a value of an unsuitable type to index the unpacked array in a PACK or UNPACK statement.

```
PROGRAM fail;
TYPE t1 = (a, b, c, d);
     t2 = 0..3;
VAR p : PACKED ARRAY [t1] OF char;
    u : ARRAY [t2] OF char;
BEGIN
   pack(u, a, p);   {a is not of type t2}
   unpack(p, u, a); {ditto}
END.


*ERROR 143 -- this string contains more than 255 characters
```

The compiler limits the maximum number of characters in a string constant to 255. It is unlikely that this message will ever be generated as it takes a fair degree of effort to generate a source file containing such a constant (which is why no sample program is given here).

```
*ERROR 144 -- string constants must contain at least two characters
```

This error is caused by a string or character constant which contains no characters, i.e. two consecutive quotes. The Pascal standard states that one character between quotes is a constant of type CHAR and that string constants must have two or more characters in them.

```
PROGRAM fail(output);
BEGIN
   writeln('');    {'' is a null string}
END.
```

```
*ERROR 145 -- a string constant is required here
```

The keywords INCLUDE and ALIAS must be followed by a sequence of characters enclosed in single quotes. INCLUDE and ALIAS are extensions to standard Pascal.

```
PROGRAM fail(output);
IMPORT PROCEDURE process ALIAS jim(x:integer);
BEGIN
   process(123);
END.
```

```
*ERROR 146 -- a string constant may not extend over more
       than one line
```

This error is issued when a line-break is found inside a string constant. It is usually caused by omitting the closing quote of the string.

```
PROGRAM fail(output);
BEGIN
   Writeln('the final quote is missing);
END.
```

```
*ERROR 147 -- strings of differing lengths may not be compared
```

The standard does not permit strings to be compared unless they contain the same number of characters.

```
PROGRAM fail(output);
BEGIN
   IF 'cat' > 'mouse' THEN writeln('bigger');
END.
```

```
*ERROR 159 -- an underline is required between a machine-code
              mnemonic and its operands
```

Assembly language statements embedded in Pascal programs must be written with a underline character between the instruction mnemonic and the first operand. Machine-code is an extension to standard Pascal.

```
PROGRAM fail;   {this needs the EXTEND option}
VAR x, y : integer;
BEGIN
   *MOVE x,y;
END.


*ERROR 160 -- only STATIC, IMPORT or EXPORT variables may be declared here
```

When a MODULE is being defined only objects which have some existence when the module is not being executed may be declared at the outermost level. Modules are an extension to standard Pascal.

```
MODULE fail;  {this needs the EXTEND option}
VAR j,k,l : integer;
END.
```

Appendix B

Fatal compilation errors

```
*FATAL ERROR 148 -- the compiler cannot complete the compilation
                    as $INCLUDE files are nested too deeply
```

This implementation of Pascal will limit the depth to which an included file may contain an included file, and so on. This error indicates that that limit has been exceeded. The usual solution is to expand the deepest include files in place.

```
*FATAL ERROR 149 -- the compiler cannot complete the compilation
                    as the program is too big to be compiled
```

This error indicates that the program has managed to fill all of the compiler's tables. The only solution is to provide more memory for the compiler to use.

```
*FATAL ERROR 150 -- the compiler cannot complete the compilation
                    as faults are occurring at too great a rate
```

During the compilation of a program the compiler keeps track of the rate at which errors are being detected. If this rate becomes too great the compilation is abandoned as it is likely that more error messages will only be confusing. The common cause of this is attempting to compile something which is not a Pascal source program, for example a data file or an object file.

```
*FATAL ERROR 151 -- the compiler cannot complete the compilation as
                    the end of the program has been found too soon
```

This fatal error is caused by the compiler discovering either the physical end of the source file or the sequence 'END.' when it was expecting more Pascal statements. The common cause for this is the omission of END statements either explicitly or as a side-effect of previous errors which confused the compiler.

```
*FATAL ERROR 152 -- the compiler cannot complete the compilation
                    as the program contains too many identifiers
```

This message is issued when the compiler has filled the tables used to hold information about all the named objects in the program. The solution to the problem is to alter the program so that more use is made of local variables rather than global ones. Local variables only require space in the compiler while the block which contains them is being processed. All the space used is made available for re-use once the final END of the block is reached.

```
*FATAL ERROR 153 -- the compiler cannot complete the compilation
       as this expression is too complicated to be analysed
```

This error is generated when the space reserved by the compiler for processing expressions has been filled. Such an error should only ever be generated by programs with enormous expressions, hundreds of additions in one expression for example. The solution is to break the expression into two or more simpler expressions. For example:

```
rewrite:   A := B+C+D+E+F+G+.........+X+Y+Z;

as:       T1 := B+C+D+E+F+....+M+N;
           A := T1+O+P+Q+.......+X+Y+Z;
```

```
*FATAL ERROR 154 -- the compiler cannot complete the compilation
       as an internal compiler error has occurred.
       please submit an error report
```

This error indicates that the compiler has got itself hopelessly lost. If any other errors have been detected then correct them and try the compilation again. If it still generates this message then a Fault Report Form should be submitted to Acorn Scientific (see beginning of this manual for address).

```
*FATAL ERROR 155 -- the compiler cannot complete the compilation
       as there are too many long identifiers
```

This error indicates that the compiler's dictionary, in which it holds the text of identifiers, has become full. Try replacing some of the longest identifiers with shorter ones.

```
*FATAL ERROR 156 -- the compiler cannot complete the compilation
       as there is not enough store to compile this program
```

This message means that the compiler cannot get the memory it needs to compile the program. The only solutions are either to simplify the program or to get more memory.

```
*FATAL ERROR 157 -- the compiler cannot complete the compilation
                    as the file <name> cannot be included
```

The program has indicated that the file <name> is to be included but the compiler cannot read the file, either because it does not exist or because the file is protected in some way.

Appendix C

Execution errors

```
*Execution error -- user termination
```

This systems permits the user to force an executing program to stop with this message in order to provide diagnostic information, for example when a program goes into an infinite loop.

```
*Execution error -- real value too large
```

This message indicates that an real (floating-point) operation has resulted in a value which is too large to be handled by the normal real hardware.

```
PROGRAM error(output);
VAR n     : real;
    times : integer;
BEGIN
   n := 1.0;
   FOR times := 1 TO 100 DO
      n := SQR(n);          {bound to get too big}
   writeln('Error not detected');
END.
```

```
*Execution error -- attempted division by zero
```

The program has attempted to divide one quantity by zero.

```
PROGRAM error(output);
VAR j, k : integer;
BEGIN
   j := 123;
   k := 0;
   writeln('123/0 = ', j/k);    {wrong}
   writeln('Error not detected');
END.
```

```
*Execution error -- x MOD <number> is not permitted
```

The definition of the operator MOD states that it shall be an error if in an expression of the form: i MOD j the value of j is less than or equal to zero.

```
PROGRAM error(output);
VAR i, j, k : integer;
BEGIN
   i := 123;
   j := -1;
   k := i MOD j;          {wrong}
   writeln('Error not detected');
END.
```

```
*Execution error -- zero or negative argument for logarithm
```

The mathematical function LN (logarithm to base 'e') is not defined for arguments less than or equal to zero.

```
PROGRAM error(output);
VAR j : integer;
    l : real;
BEGIN
   FOR j := 5 DOWNTO -5 DO
      l := LN(j);              {should fail}
   writeln('Error not detected');
END.
```

```
*Execution error -- negative argument for square root
```

The mathemetical function SQRT (square root) is not defined for arguments less than zero.

```
PROGRAM error(output);
VAR j : integer;
    r : real;
BEGIN
  FOR j := 5 DOWNTO -5 DO
      r := SQRT(j);            {should fail}
   writeln('Error not detected');
END.
```

```
*Execution error -- significance lost
```

The trigonometric functions are most accurate when their arguments are in the primary range (about -PI..PI). The further the argument gets from this range the less accurate is the result. The error indicates that the argument was too far from the primary range that the result is likely to be so inaccurate as to be meaningless.

```
PROGRAM error(output);
VAR s, a : real;
    n    : integer;
BEGIN
   a := 1;
   FOR n := 1 TO 1000 DO BEGIN
      a := 2*a;
      s := SIN(a);              {should fail eventually}
   END;
  writeln('Error not detected');
END.
```

```
*Execution error -- not enough store
```

The program has requested more working memory than is available. The common causes for this error are either declaring enormous arrays or calling procedures or functions recursively without a suitable termination condition. Note that arrays of arrays can use up space very quickly.

```
PROGRAM error(output);

   {Note that this program could just blow up if}
   {the compiler's check for running out of store}
   {fails to catch the error}

PROCEDURE recurse(n : integer);
BEGIN
   IF n > 0 THEN recurse(n+1);
END;

BEGIN
   recurse(1);  {this will never come back}
END.
```

```
*Execution error -- out of range
```

This message indicates that a value is outside the range demanded by its use. This is commonly caused by using an incorrect value as an index into an array.

```
PROGRAM error(output);
TYPE small = 1..5;
VAR  a : array [small] of integer;
     j : integer;
BEGIN
   FOR j := 1   TO   5 DO a[j] := SQR(j);   {fill the array}
   FOR j := 5 DOWNTO 0 DO writeln(a[j]);    {fails when j = 0}
   writeln('Error not detected');
END.
```

```
*Execution error -- unassigned variable
```

This message is caused by attempting to use the value of a variable before any value has been put into it. When a variable is declared its contents are undefined and remain so until a value is assigned.

```
PROGRAM error(output);
VAR a : ARRAY [1..6] OF integer;
    j : integer;
    c : char;

   {Note that the variable c is left unassigned throughout}
   {the execution of this program, but as its value is}
   {not used this cannot cause an error}

BEGIN
   FOR j := 1 TO 5 DO  a[j] := -j;
   FOR j := 1 TO 6 DO  writeln(a[j]);   {a[6] is unassigned}
   writeln('Error not detected');
END.
```

```
*Execution error -- input/output error -- <reason>
```

This message is detected by the operating system and indicates that something has gone wrong with an input or an output operation. The message <reason> should give further information about the cause of the error.

```
*Execution error -- NIL pointer used
```

This message indicates that an attempt has been made to follow a pointer which is currently NIL. As this means that the pointer is not pointing at anything the operation would be meaningless.

```
PROGRAM error(output);
TYPE pi = ↑integer;
VAR a : pi;
BEGIN
   a := NIL;
   a↑ := 123;   {wrong}
   writeln('Error not detected');
END.
```

```
*Execution error -- disposing NIL pointer
```

This error is the result of calling DISPOSE on a pointer which has the value NIL.

```
PROGRAM error(output);
VAR pi : ↑real;
BEGIN
   pi := NIL;
   Dispose(pi);
   writeln('Error not detected');
END.
```

```
*Execution error -- reference to DISPOSED object
```

This message indicates that a pointer has been left pointing to the remains of an object that has been disposed. This is usually the result of copying a pointer variable and using the copy after DISPOSE has been applied to the original.

```
PROGRAM error(output);
TYPE pi = ↑integer;
VAR original, copy : pi;
BEGIN
   NEW(original);
   original↑ := 123;    {valid}
```

```
    copy := original;     {copy it}
    DISPOSE(original)
    copy↑ := 0;           {there's nothing there}
    writeln('Error not detected');
END.


*Execution error -- missing case
```

This message indicates that a CASE statement has attempted to select a case which has not been specified. Note that the compiler will often warn that this is a possibility while the program is being compiled.

```
PROGRAM error(output);
TYPE tiny : 1..3;
VAR n : tiny;
BEGIN
   FOR n := 1 TO 3 DO BEGIN
      CASE n OF                {fails when n = 2}
         1: writeln('one');
         3: writeln('three');
      END;                     {no case 2}
   END;
   writeln('Error not detected');
END.


*Execution error -- GET after EOF
```

This error indicates that the procedure GET has been called when the file it is operating on is at end-of-file (EOF) and so there is no more data available. The call on GET could either be explicit or implied by a reference to READ or READLN.

```
PROGRAM error(output);
VAR f : text;
    c : char;
BEGIN
   rewrite(f);     {make it empty}
   reset(f);       {prepare to read it}
   read(c);        {nothing to read}
   writeln('Error not detected');
END.
```

```
*Execution error -- reset on non-existent file -- <filename>
```

This message is the result of calling RESET on a file which does not exist. The most common cause is declaring a file variable and applying reset to it before anything has been sent to it.

```
PROGRAM error(output);
VAR f : text;
BEGIN
   reset(f);          {it doesn't exist yet}
   writeln('Error not detected');
END.
```

```
*Execution error -- rewrite fails -- <reason>
```

This message is issued when the host operating system cannot open a file for output. One of the more common causes of this is an attempt to open more files at once than is permitted. <reason> should be an explanatory message from the operating system.

```
*Execution error -- no association for <name>
```

This message is generated when an attempt is made to execute a program without specifying the actual file to be used when the program parameter <name> is referenced.

```
*Execution error -- RESET required before read access
```

This error is caused by an attempt to read from an output file. Before the data in an output file may be accessed the file must be made an input file by calling the procedure RESET.

```
PROGRAM error(output);
VAR f : text;
    c : char;
BEGIN
   rewrite(f);    {it's an output file}
   read(f, c);    {wrong}
END.
```

```
*Execution error -- REWRITE required before write access
```

This error is caused by an attempt to write to an input file. Before the data in an input file may be accessed the file must be made an output file by calling the procedure REWRITE.

```
PROGRAM error(output);
VAR f : text;
BEGIN
   rewrite(f);                    {create it}
   writeln(f, 'this is a line');  {fill it}
   reset(f);                      {make it input}
   writeln(f, 'oops');            {wrong}
   writeln('Error not detected');
END.
```

```
*Execution error -- EOLN invalid at EOF
```

This error is caused by calling the boolean function EOLN when the file is at end-of-file (EOF). The function EOF(file) should usually be tested before EOLN to avoid this error.

```
PROGRAM error(output)
VAR f : text;
BEGIN
   rewrite(f);
   writeln('one line');
   reset(f)
   readln;                {nothing left in f now}
   IF EOLN(f) then writeln('end of line');
   writeln('Error not detected');
END.
```

```
*Execution error -- invalid number syntax
```

This error is caused by an attempt to read a numerical value (integer or real) from a file if the input cannot be interpreted as a number. Note that the Pascal standard only permits spaces and end-of-lines to be skipped before a valid number is found. In particular a comma will not be skipped and will invoke this error.

```
PROGRAM error(output);
VAR f    : text;
    n, m : integer;
BEGIN
   rewrite(f);
   writeln(f, '123, 456');
   reset(f)
   read(f, n);    {will input the value 123}
                  {leaving the comma as the next}
                  {character to be input}
   read(f, m);    {the comma makes this fail}
   writeln('Error not detected');
END.
```

```
*Execution error -- <number> is not an acceptable field width
```

This error is caused by attempting to specify an output field width with a value which is less than one.

```
PROGRAM error(output);
VAR f : text;
    j : integer;
BEGIN
   rewrite(f);
   FOR j := 3 DOWNTO -3 DO
      writeln(f, 12345:j);  {fails when j=0}
   writeln('Error not detected');
END.
```

```
*Execution error -- cannot extend heap
```

The heap is the area of memory in which the compiler manages the storage claimed and released by calls on NEW and DISPOSE. This error is issued when the area is full and no more memory can be found. The common cause of this error is calling NEW repeatedly without releasing unwanted space by DISPOSE.

```
PROGRAM error(output);
TYPE bigrec = RECORD a : ARRAY [1..100] OF integer END;
VAR  pr : ▮bigrec;
```

```
BEGIN
   REPEAT
      NEW(pr);
   UNTIL false;    {loops forever}
   {there is no way out of the loop}
END.


*Execution error -- corrupt program
```

This message is usually only generated from programs which have been compiled with all of the run-time checks disabled. It is the result of parts of the program being overwritten as the result of undetected errors elsewhere in the program. If this occurs try recompiling the program with all checks turned on (or the default state). If the fault persists please submit a Fault Report Form to the address at the beginning of this manual.

Index

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200