FORTRAN 77

REFERENCE
MANUAL

FORTRAN 77

PART NO 0410, 008
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 41 2 Acorn Scientific

Contents

1 Introduction to 32000 FORTRAN 77

FORTRAN has long been regarded as the programming language most suited to scientific and numeric applications. FORTRAN 77 is the latest standardised version of the language, and this has been used in the production of Acorn 32000 FORTRAN 77. This manual describes the use of Acorn 32000 FORTRAN 77 running under the Panos operating system; note that it is not a tutorial.

The Acorn 32000 FORTRAN 77 compiler has been fully validated in conformance with the American National Standard Programming Language FORTRAN X3.9 -1978 (ANS FORTRAN). Detailed language specifications are given in the publication *American National Standard Programming Language FORTRAN, X3.9-1978*, which is available from the British Standards Institute.

From now on, unless otherwise stated, or made obvious from the context, 'FORTRAN 77' is taken to mean the implementation of FORTRAN 77 on Acorn NS32000 based computers under the Panos operating system.

1.1 Installation

The language is provided on an Acorn DFS format disc, and before use it must be installed as described in the appropriate *User Guide* onto either the DFS, the ADFS (typically for use with a Winchester disc), or the NFS (in conjunction with the Econet). Note that the installation procedures must be followed even if you are installing FORTRAN 77 onto floppy disc, despite the fact that you have received it on this medium. The *User Guide* also contains some simple examples in the use of the compiler, and it is recommended that you read the relevant chapter before continuing. See also the *Panos Guide to Operations* which provides information about the Panos operating system at the level of the user interface, such as the use of utilities, including the linker.

2 The Compiler

The FORTRAN 77 compiler is made up of two parts: a front end which checks that the source code conforms to the standard, and a code generator which creates the equivalent machine code program. There are a number of arguments which can be issued to the compiler to give extra control over the compilation, and allow the compilation options to be used.

The command 'f77' is, in fact, a command file which runs the two parts of the compiler in sequence, and so compiles the program without the need for the user to give two separate commands. It also informs the user how to find out help information from the front end and code generator, and accepts the compiler arguments. The *User Guide* contains more details about the 'f77' command, and the *Panos Guide to Operations* provides instructions for creating or modifying command files.

2.1 Compilation arguments

The behaviour of the compiler can be modified through the use of compilation arguments. These can be used to specify input and output files, listings, identification, and also the compilation options, which are a type of argument specifying 'lower-level' compiler activity.

Any number of arguments can be given, and in any order. The form of the command line is as follows:

```
-> f77 [-source] name { [-list [name]] [-aof name]
       [-error name] [-opt options] [-map name]
       [-identify] [-help] }
```

where 'name' stands for a user-supplied file name, and 'options' represents a list of one or more compiler options. Braces enclose optional items.

-source name

The source file is the only argument which is not optional (although the keyword '-source' is). It specifies the name of the file which contains the code to be compiled. If the supplied name has no extension, then '-f77' will automatically be appended. See the *Panos Guide to Operations* for details about the Acorn file extension conventions.

-list name

A listing of the compiled program along with line numbers of the source is generated when this argument is given. This listing can be sent to a file or device specified in the argument (e.g. printer:). If no file name or device is specified, then the compiler sends the listing to 'name-lis'; a file whose name is based on the source file name, e.g. a file called Fprog-f77 will have a listing file called Fprog-lis. See figure 1 for a demonstration of this command.

-opt options

Several options are accepted by the compiler. These are given in the opt argument. The options available are listed below under the heading 'Compilation Options'.

-error name

Compiler error messages are sent to the vdu by default, but may be re-directed using the error argument to a specified file or device.

-aof name

The Acorn Object Format output file of the compiler is given a name based upon the source file name by default i.e. a file called 'fort1-f77' will be given the aof name 'fort1-aof'. The -aof argument can be used to specify an alternative name.

-map name

A storage map of the compilation is produced by the code generator and sent to the device or filename specified. This is given the file extension '-map' by default.

-identify

The identify argument requests compiler identification.

-help

This argument, if given to the command file 'f77', tells you how to obtain help information from the front end and code generator. The actual commands neccessary are f77fe -help and f77cg -help.

Example compiler commands

A. The minimal command

```
f77 Fprog
```

This command will compile the source program 'Fprog-f77'; default settings are used throughout, i.e. the intermediate file is called 'fcode-tmp', the aof file is called 'Fprog-aof', no map or listing is produced, error messages are directed to the vdu, and default compilation options are assumed (see section below).

B. Specifying the input and output files

```
f77 -source Fprog -fcode $.junk.code
-aof $.Afiles.Fprog -list Proglist -error printer:
```

The optional '-source' argument is supplied, and the source program 'Fprog-f77' is compiled with the intermediate file being called '$.junk.code', the aof file named as '$.Afiles.Fprog-aof', a listing is sent to the file 'Proglist-lis', and all error messages resulting from the compilation are sent to the printer.

C. Using some options and requesting a map:

```
f77 Fprog -opt +tW0 -map dfs::0.FProg
```

The program 'Fprog-f77' is compiled, with tracing specified in the code, warning messages inhibited, and a storage map sent to the file 'dfs::0.Fprog-map'.

2.2 Compilation options

The -opt argument is followed by a list of compilation options (in upper or lower case). The options 'B', 'H', 'T', and '6', are enabled or disabled by preceding them with + or -. The options 'X', 'L', and 'W' must be followed by a number. The default for the full set of options is:

```
L1W2X0 -BHT6
```

This means that code generator line numbering is set to level 1; level 2 warning messages are given; there is no cross-referencing output, no bound checking, and Hollerith constants are not allowed; tracing and FORTRAN 66 are disabled.

B

Causes the compiler to generate bounds checking code. Array or substring subscripts out of range will cause run-time errors to be reported in programs compiled with this option.

H

When enabled, this option allows Hollerith constants to be used in DATA statements to initialise non-CHARACTER variables (e.g. INTEGER).

L n

This option is followed by a number which indicates the level of line numbering included in the code for backtrace purposes (see chapter 5). The levels available are:

0	no line numbering
1	numbers lines containing subprogram CALLS
2	statements which can cause a run-time exception (e.g. divide by zero);
>2	numbers every line

Higher levels cause more code to be generated. If a hardware exception occurs in a module compiled with level 1, the backtrace system will not be able to determine the exact line number; instead, a range of numbers will be given (e.g. 100/106). The error will lie in this range.

T

This causes the compiler to plant tracing code in the output file (see chapter 5).

6

This option allows FORTRAN 66 features to be used; if enabled, it implies the 'H' option.

W n

This sets the warning message level. A following digit of 0-4 is interpreted from the zero level, as 'suppress all warnings' to 'print all warnings' (level 4). See chapter 5 for more details.

X n

This is followed by a cross-reference listing width (of 18 or more for maximum legibility). A value of zero suppresses cross-referencing. The upper limit depends upon the device to which the listing is being sent (e.g. printer). Cross-reference information is given immediately after the END statement of a program unit. For each name, the type is given, together with the lines on which it is referenced. For each statement label, the type (executable or non-executable) and line number of the statement is given, as well as the lines on which the label is referenced.

3 Extensions to the standard

32000 FORTRAN 77 offers several enhancements to the standard which are described in this chapter. Further extensions concerning input/output are described in chapter 4.

3.1 Hexadecimal constants

32000 FORTRAN 77 allows hexadecimal constants to be used wherever an ordinary constant is allowed. A hexadecimal constant is of the form:

?<type><digits>

<type> is a letter, specifying the type of the constant. It must be one of I, R, D, C, L or H (for INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL and CHARACTER respectively).

The <type> letter is followed by hexadecimal <digits> (0-9, A-F). There must always be an even number of digits (i.e. an exact number of bytes).

The bytes in a CHARACTER hexadecimal constant are given in the order in which they are to appear in store; with other constants, the most significant byte is given first. If the type of the constant is REAL, DOUBLE PRECISION or COMPLEX, the number of bytes must match the size of the item in store (4 or 8); for INTEGER and LOGICAL constants, there may be fewer bytes. For example:

```
CHARACTER WINDOW*(*)
PARAMETER (WINDOW = ?H1C05141E0C)
J = ?I1234
```

Here, 'window' consists of the bytes 1C 05 14 1E 0C, and 'j' is set to the decimal value 4660.

3.2 FORTRAN 66 option

Quoting the +6 option in the command line (see chapter 2) specifies that the compiler will be in FORTRAN 66 mode. When this option is enabled, the action of FORTRAN changes as follows:

- DO loops will always execute at least once.
- Hollerith constants (nH) are allowed in DATA and CALL statements, and quoted constants are not CHARACTER type.

When the FORTRAN 66 switch is used, both Hollerith and quoted constants in CALLS and DATA are treated in the same way - they are not of CHARACTER type. The option is provided for use with FORTRAN 66 programs which store character information in numeric data types. For example, the following calls will have identical effects at run time if the FORTRAN 66 switch is used:

```
CALL JIM('ABCD') AND
CALL JIM(4HABCD)
```

Run-time FORMATs may be non-CHARACTER array names if the +6 option is quoted. For example:

```
DOUBLE PRECISION D(3),NUM
DATA D(1),D(3)/8H(1X,D20.,5H,I5/)/
..
DATA NUM /2H10/
..
WRITE (6,D) 2.3D0,10
..
```

This facility was introduced for the compilation of FORTRAN 66 programs. It is strongly recommended that new programs use CHARACTER formats.

3.3 Naming

In 32000 FORTRAN 77, all lower case letters (except in FORMAT s and character constants) are converted into upper case upon reading the source. Thus all statements, identifiers etc. may be in lower case. Names may be up to 255 characters long, though a warning message is given if names longer than the 'standard' limit of 6 characters are used. It is worth noting, to save confusion, that there is no limit on the length of CHARACTER values.

4 Input/output

This chapter describes how 32000 FORTRAN 77 input and output functions are implemented and how this affects programs.

4.1 Unit numbers and files

A FORTRAN unit number is a means of referring to a file. In the Panos system, unit numbers in the range 1 to 60 may be used, as well as the two * units for the keyboard and screen. Note that there is a limit of 5 files that can be open simultaneously on the DFS, and 10 on the AD/NFS.

A unit number may be connected to an external file either by means of an OPEN statement or by assignments on the command line when the program is run. If an OPEN statement with the FILE= specifier is used, then the unit is connected to the given filename, otherwise, the command line parameters are scanned.

The format of the command line is:

command {file*} {unit=file*}

i.e. an optional list of filenames followed by an optional list of assignments of a particular unit to a named file. The initial series of 'unkeyed' filenames are connected to units 1, 2, 3... Each 'keyed' file is connected to the given unit number. All 'unkeyed' definitions must precede any 'keyed' definitions.

Examples are:

```
-> prog abc def
```

This associates the file 'abc' with unit 1 and 'def' with unit 2.

```
-> prog 10=file-txt
```

This associates the file 'file-txt' with unit 10.

```
-> prog data-tmp 32=data 3=x
```

This associates 'data-tmp' with unit 1, 'data' with unit 32, and 'x' with unit 3.

The two * units always refer to the initially selected input/output streams. Any units which are not connected to a file in an OPEN statement or command line assignment also refer to these streams.

All files are closed automatically when a program terminates.

When writing to a sequential formatted file, a distinction is made between files which are to be 'printed' and those which are not. In the former case, the first character of each record is taken as a carriage control, and does not form part of the data in the record. Since any file may eventually be printed, some means is required in FORTRAN for specifying whether a given unit is to be treated as a printer. This may be done in one of two ways:

The two * units, and all units in the range 50-60, assume 'printer' output by default.

Quoting `FORM='PRINTER'` in an OPEN statement for the unit causes 'printer' output to be assumed for that unit (N.B. this is an extension to the standard).

Note that 'printer' output does not imply output to any physical printer which may be connected to the machine.

The carriage control characters which are recognised, and their representation in files, are described below.

4.2 Sequential files

4.2.1 Opens and closes

An OPEN statement for a sequential file does not specify the direction of transfer that is required, so the actual system open operation cannot be done until the first READ or WRITE statement following the OPEN. For this reason, an OPEN statement which refers to a file which does not exist will not fail - the error will occur when a READ or WRITE is attempted, but may then be trapped by use of an ERR= specifier.

A sequential unit may be used without an explicit OPEN operation, in which case the file is actually 'opened' on the first READ or WRITE which refers to the unit.

The following subroutine is an example of the use of OPEN and ERR=. The routine copies a named file to the terminal, using unit 10.

```
      SUBROUTINE COPY(FILE)
      CHARACTER FILE*(*), LINE*72
      OPEN (10, FILE=FILE, ERR=100)
    1 READ(10, '(A)', END=100, ERR=100) LINE
      PRINT '(1X, A)', LINE
      GOTO 1
  100 CLOSE (10)
      END
```

4.2.2 Formatted IO

Formatted (and list-directed) reads and writes are permitted on all files.

A formatted READ statement causes one or more records to be read from the file or terminal. All input records are assumed to be extended indefinitely with spaces, so that an input format may refer to more characters than are actually present in the record. Input from the terminal uses the normal line-editing conventions (including cursor copying). CTRL - D is treated as 'end of file', which may be trapped by an END= specifier in a READ statement.

For file input the characters carriage return (CTRL-M) and line feed (CTRL-J) are each recognised as record terminators. Form feed (CTRL-L) characters are ignored. If the record contains more than 512 data characters, the rest are ignored.

When writing a record to a file or terminal, the carriage control characters(s) are output first, followed by the data in the record. Trailing spaces are removed from all output records.

The following carriage control characters are recognised:

space	Performs a line feed (LF)
0	Performs LF/LF (extra blank line)
1	Performs CR/FF (newpage)
+	Performs CR (overprint)
*	No action taken

The initial LF (space/0) or CR (1/+) is not output before the first record in the file.

When writing to a 'non-printer' file, the effect is the same as for a space carriage control. An unrecognised control character is treated as space.

The * carriage control (an extension) may be of use when writing control codes to the VDU driver.

When a file is closed, a line feed character is output if the final record contained any data characters. This is done automatically for all open files when a program terminates normally.

A write to a terminal file causes the record to be output to the screen immediately, but the following carriage control characters will not be output until the next WRITE or PRINT statement. Therefore, a statement like:

```
PRINT *, 'Type an integer:'
```

may be used to output a 'prompt' to the terminal.

The following example program illustrates interaction with a terminal file:

```
1 PRINT *, '?'
  READ (*, *, END=3) I
  WRITE(*, 2) I, I*I
2 FORMAT('+', 2I10)
  GOTO 1
3 END
```

The + carriage control in the output format is used to prevent a blank line occurring between the input line and the response. If a prompt string is not used, it will be necessary to output an extra record after the response, to move the cursor to the next line. This may be done by a '/' at the end of the format:

```
2 format('+', 2i10/)
```

The CHAR function may be used to construct bytes for output as VDU control codes. For example, the following statements will switch the screen to MODE 3 on your machine:

```
      WRITE(52,3), CHAR(22), CHAR(3)
3     FORMAT(1H*,2A)
```

During formatted input of numeric values, blanks are either ignored, or treated as zeros, depending on the use of the BZ and BN format specifiers, and the BLANK status of the unit. All 'preconnected' units (i.e. those opened without explicit use of OPEN) have BLANK = ZERO as the default status. Any unit connected by an OPEN statement has BLANK = NULL as

the default. The difference in the defaults was introduced for compatibility with FORTRAN 66 and the FORTRAN 77 subset language (in FORTRAN 66, blanks are always treated as zeros).

Note that to send control codes, the file must be connected to the Panos stream rawvdu:, for example:

```
-> prog 52=rawvdu:
```

or in the program:

```
OPEN(52,FILE='RAWVDU:')
```

The screen should be in a graphics mode before running the program. It is possible to change mode from within a 32000 FORTRAN 77 program. A routine which will do this can be found in the FORTRAN 77 graphics library on the 'Welcome' disc.

4.2.3 Unformatted IO

Unformatted reads and writes are permitted on disc files only. Unformatted and formatted operations may not be mixed on any unit, unless the unit is closed and reopened.

Each unformatted WRITE statement writes a single 'record' to the file. The record may be read back later by any READ which quotes the same number, or fewer, variables. For example, in:

```
WRITE(1) 1, 2, 3, 4, 5
WRITE(1) 6, 7, 8
REWIND 1
READ(1) I
READ(1) J
```

'i' is to 1 and 'j' to 6. The first record contains 20 bytes of data, and the second 12 bytes.

The desired effect could be achieved by padding all unformatted records to the same length, but this would lead to wasted file space in many cases. The system includes a record length before every unformatted record when it is output, and always reads the right amount when the record is read again.

The actual format of the length is: the characters 'UF' followed by a four-byte byte count giving the number of data characters following. The UF bytes are used as a check that the file contains valid unformatted

records. For example, the two records written in the example above would contain the following bytes:
55 46 14 00 00 00
01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00
55 46 0C 00 00 00
06 00 00 00 07 00 00 00 08 00 00 00

4.3 Direct Access files

A direct access file consists of a number of records, all of the same length, which may be read and written in any order. The records are either all formatted or all unformatted.

An OPEN statement, quoting the record length, is always required when using a direct access file. The record length is measured in bytes, and formatted records are padded to this length with spaces.

A direct access file starts with six special bytes which identify it and give the record length. These bytes are the characters 'DA' followed by the record length as a four-byte value (LS byte first). It is permissible to OPEN a direct access file quoting a smaller record length than was given when the file was created.

The maximum permitted record length in a formatted direct access OPEN is 512 bytes; there is no limit for unformatted files.

If the file has been opened for updating or input, the first six bytes of the file are read and checked. The OPEN will fail if these bytes are invalid, or the specified record length is greater than the value used when the file was created.

Since it is possible both to read and write to a direct access file, the system open operation may be performed as part of the OPEN statement, rather than being delayed to the next READ or WRITE, as is the case with sequential OPENs. Therefore any errors which occur in the open may be trapped by an ERR = specifier in the OPEN statement.

Note that a direct access OPEN may refer to an existing file only if it is of the correct format.

The following is an example program which uses direct access to write and read a file on unit 42:

```
      OPEN(42, ACCESS='DIRECT', FILE='DAFILE', RECL=16,
     1 +     ERR=100, IOSTAT=IERR)
      DO 1 J = 20,1,-1
     1 WRITE(42, REC=J) J, J+1, J*J, J-1
      DO 2 J = 1,10
      READ (42, REC=J) K, L, M
     2 WRITE(*, 3) K, L, M
     3 FORMAT(1X, 3I5)
      STOP
   100 PRINT *, 'OPEN FAIL: ', IERR
      END
```

Note that unformatted records are the default for direct access files. The file 'dafile' used in the above example need not exist already, but if it does, it must be a valid direct access file with a record length not less than 16.

4.4 OPEN and CLOSE

The OPEN and CLOSE statements have been discussed above. The NEW and OLD values for the STATUS specifier in the OPEN statement are ignored.

4.5 INQUIRE

4.5.1 INQUIRE by unit

An INQUIRE by unit operation gives information on a particular unit. The EXIST specifier variable is set to .TRUE. if the unit is in the valid range. It is impossible to give accurate responses to the SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED specifiers, so 'YES' is returned if the unit is actually being used for the relevant access type, and 'UNKNOWN' is returned otherwise. Note that a unit is NAMED only if a FILE specifier was quoted in the OPEN statement for the unit; command line file assignments are not available to INQUIRE.

4.5.2 INQUIRE by file

An INQUIRE by file operation gives information on a particular filename. If the file has been quoted in an OPEN statement for a unit (and not CLOSEd), information deduced from that connection is returned (e.g. DIRECT is set to 'YES' if the file is open for direct access), and the file is assumed to exist. Otherwise, if the file exists, the EXIST reply is .TRUE. and the responses to the SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED specifiers are 'UNKNOWN'.

4.6 BACKSPACE

BACKSPACE is not implemented.

4.7 ENDFILE

The operation of ENDFILE is entirely internal to the run-time system; the only effect is to set 'end of file' status and forbid further access to the file.

4.8 REWIND

REWIND is implemented as a CLOSE followed by an OPEN. After executing a REWIND, the file is in a similar state to that arising after an OPEN statement - the system open operation is awaiting the next READ or WRITE statement.

4.9 FORMAT decoding

Format specifications are decoded in a rather more liberal manner than implied by the FORTRAN standard.

4.9.1 Lower case letters

Lower case can be used instead of upper case everywhere; cases are distinguished only in quoted strings and nH descriptors, and in the `D`, `E` and `G` edit descriptors (see below).

4.9.2 Extraneous repeat counts

Unexpected repeat counts are ignored - i.e., before `'`, `T`, `/`, `:`, `S` and `B` edit descriptors, before the sign of a `P` edit descriptor, or before a comma or closing parenthesis.

4.9.3 Edit descriptor separators

A comma may be omitted except where the omission would cause ambiguity or a change in meaning - thus it cannot be omitted between a repeatable edit descriptor (such as `I5`) and an `nH` edit descriptor (such as `11Habcdefghijk`).

4.9.4 Numeric edit descriptors

As well as the standard forms `Iw`, `Iw.m`, `Fw.d`, `Ew.d`, `Ew.dEe`, `Dw.d`, `Gw.d` and `Gw.dEe`, additional forms are: `Fw`, `Dw.dDe`, `Gw.dDe`, `Dw.dEe`, `Ew.dDe` `Zw`, and `Z`

When the exponent field width is specified, the letter used to introduce it is used in the output form (in the same case). If no exponent field width is specified, then except for `G` edit descriptors the initial character of the descriptor is used in the output form (again, in the same case).

If an exponent field width is given as zero, 2 is assumed; if on output the given exponent field width is just too small for the exponent, the character introducing the exponent field is suppressed.

The `Z` edit descriptor provides input and output of numeric data in hexadecimal form. On input, the field width must equal the number of hexadecimal digits contained in the value being read (e.g. 8 for an INTEGER). On output, the width should not be less than this value; if greater, the output is padded with leading spaces. A field width of zero implies the 'right' width; 'Z' by itself is a shorthand for 'Z0'. Currently, the bytes in a numeric value are transferred in store order (LS first) when using `Z` editing; this is inconsistent with the form of hexadecimal constants in source programs, and may be changed in the future.

4.9.5 A editing

The A edit descriptor can also handle numeric list items; the effects are as recommended in Appendix C (Hollerith) of the FORTRAN 77 standard. If the field width is zero the system will automatically use the right value for the data type being transferred (4 or 8).

It must be emphasised that this use of A editing was introduced solely to aid in the transfer of FORTRAN 66 programs - it should not be used otherwise.

4.9.6 Abbreviations and synonyms

Symbol	Abbreviation
0P	P
1X	X
T1	T
TL1	TL
TR1	TR
A0	A

4.9.7 Transfer of numeric items

The I edit descriptor can be used to transfer real and double precision values; F, E, D and G can be used to output an integer value. Note that the external form of a value that is to be transferred to an INTEGER list item must not have a fractional part or a negative exponent.

5 Errors and debugging

In most cases, mistakes in a program are trapped, and indication is given as to the likely cause of the problem via error messages. Errors can be detected by the compiler and by the run-time library. An example of a fault which is not caught by the compiler, but by the FORTRAN run-time library, is attempting to divide by zero. More usually, error messages are sent from the compiler. This may also generate warning messages, which indicate to the programmer that the program may not behave as anticipated, for example, using but not declaring a variable.

5.1 Front end error messages

As mentioned earlier in chapter 2, the compiler is in two parts. Errors trapped by the front end are of a different type from those reported by the code generator. Front end error messages are short, obvious statements indicating that the compiler has spotted an unacceptable syntactic mistake. Since these messages are self-explanatory, they are not enunciated in great detail here, but listed in Appendix A. These are divided into two classes.

Class 1 errors cause the front end to abandon compilation of the current statement. The statement is printed as part of the error message, together with the number of the line on which the fault appeared, an 'error number', and a description of the error itself. Thus, if line 211 contained the faulty FORTRAN statement :

```
100         erroneous
```

then the message produced might be:

```
211   100     erroneous
 L 211 -----------?
 Error (code 2311); Statement not recognised
```

Class 2 errors may be less obvious in their report of a fault, and do not always refer to the line which contains the code which instigated the error. Thus information about missing labels is given at the end of the program unit, rather than where the non-existent label was called.

The reason that the distinction between these two types of error message has been made is to reinforce the notion that errors do not necessarily occur at the line where the message is given. Careful thought and a little imagination are often needed in order to pinpoint the cause of some persistent error messages.

A demonstration of a front end error message can be found in figure 1. This shows a FORTRAN compilation which has taken place from within the Panos editor. The source code can be seen in the background, with the compilation command in a command line window, and the listing of the compilation in the lower window.

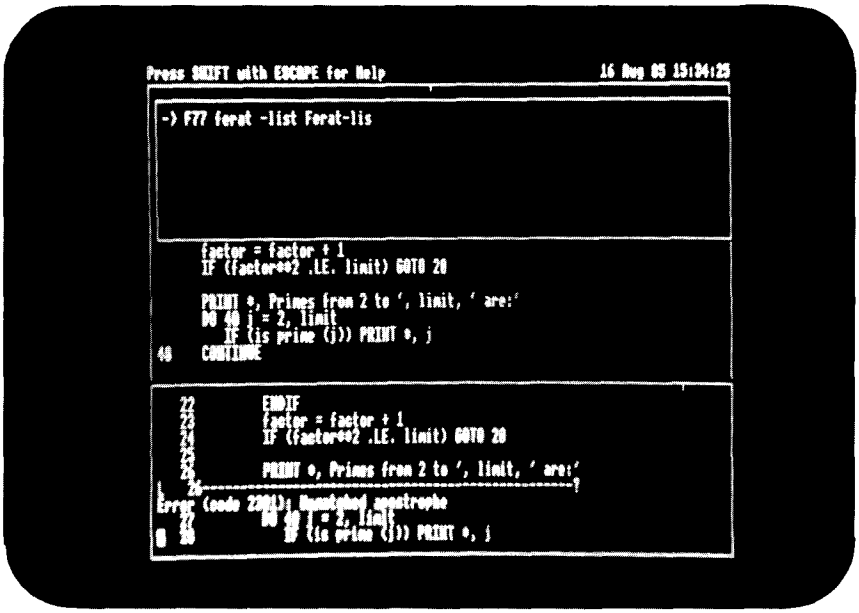


Figure 1 A FORTRAN Compilation within the editor

5.2 Warning messages

The 'W' compilation option enables the compiler to give advice in the form of warnings. See chapter 2 for more details on the use of this compilation option. These warning messages are graded upon their severity from 1 (the

most serious) to 4. They are useful in detecting areas which may cause the program to behave in unexpected ways.

Level 1 is the most serious, indicating faults such as having a statement that cannot be reached because it is unlabelled and follows a jump.
Level 2 flags the use of extensions to standard FORTRAN 77 that are a potential source of trouble (e.g. when moving software to another machine).
Levels 3 and 4 are used to indicate items that are legal but in poor style, and thus possibly mistakes.

Examples of warning messages

Level 1
- Last statement not END
- Blank statement - treated as comment
- PRINT treated as WRITE
- WRITE treated as PRINT
- Statement cannot be executed

N.B. Warning 2253 (blank statement) is produced only when the FORTRAN 66 option is set.

Level 2
- Program name omitted
- Long name '<name>'
- Non-standard hexadecimal constant

Level 3
- '<name>' typed implicitly

5.3 Code generator error messages

Certain compile-time errors cannot be detected by the front end, but are reported by the code generator. As these are not always as explicit as front end error messages, they are listed in Appendix B with a brief explanation of their most likely meaning. The same caveat applies to the interpretation of code generator error messages as applies to that of some front end error messages. The error which is reported and its line number may not directly correspond to an error in the program. For example, a real constant may be given that is too large, resulting in an error message each time the constant

is used, despite the fact that the statements which are using the constant appear to be legal. Quite often, one error may 'spark off' the detection of many others later on in the program. See Appendix B for a list of code-generator error messages.

5.4 Code generator limits

The code generator has certain internal limits on the complexity of each program unit. These are:

code size	128K bytes
number of labels	4096
number of local variables	8192
number of constants	8192
number of COMMON blocks	2048
number of external symbols	2048

These limits should never be exceeded in practice; it is likely that the code generator will run out of store before this happens.

5.5 Run-time errors

Sometimes, a program compiles correctly, links without a problem, and yet when an attempt is made to run the program, an error message is produced. These error messages come from the FORTRAN run-time library and take the following form:

```
++++ ERROR N: text
```

followed by a backtrace.

'N' is an error number and 'text' is a sentence describing the error. A backtrace is, as the name implies, a re-tracing of the steps which the FORTRAN run-time library has taken in attempting to run the program. Each line of the backtrace output gives the name of a program unit, the addresses of the corresponding module table and data area (static base), and the offset of the call with the program unit and the line number. The data area address may be used in conjunction with the storage map produced by the code generator to examine the values of local variables. The addresses

and link offset are given in decimal. Note that a name in a backtrace refers to the main entry point of the program unit, and so may not be the actual name used in a call.

Example run-time error message and backtrace

```
++++ ERROR 1000: operand negative in SQRT

Routine MOD    data area link   line

F_INIT  608     7356     255
F_SQRT  592     7292      41
DEF     544     7232      29    106
ABC     528     7172      16    210
F_MAIN  512     7168      18     99
```

In this example, the main program (with default name) has called ABC, which has called DEF, which has called SQRT (the name shown is the internal name for the instrinsic function SQRT). The final routine (F_INIT) is the main error handler.

The call to ABC in the main program was on line 99; the call in ABC to DEF was on line 210 etc. The appearance of line numbers in the backtrace is controlled by the compiler 'L' option. Level 1 line-numbering is the default case. See chapter 2 for details about compilation options.

The FORTRAN 77 library passes back to Panos a negative return code if a run-time error occurs. This may affect command sequences which run FORTRAN 77 programs. See figure 2, which shows a run-time error followed by a backtrace and a Panos return code.

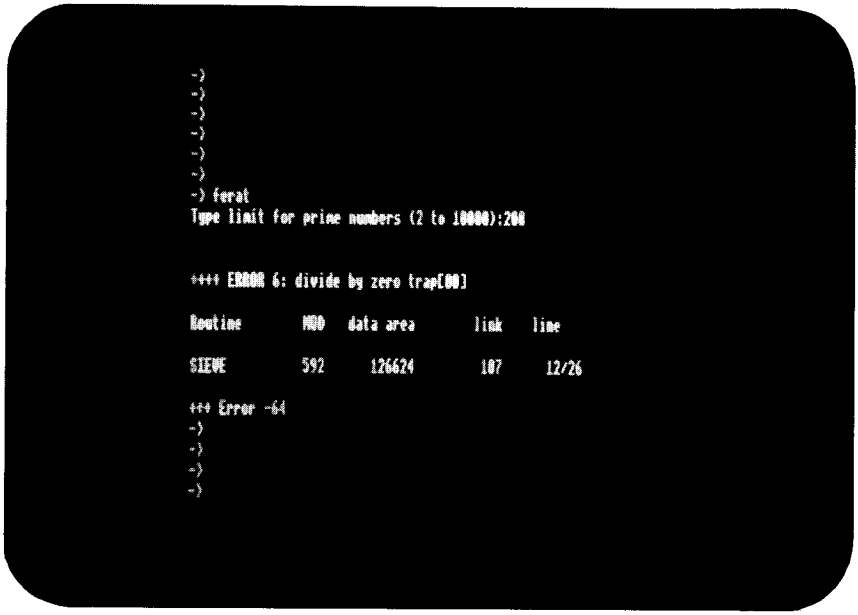


Figure 2 A Run-time error

5.5.1 Code 1000 errors

There are a number of simple run-time errors producing error messages which have an error number of 1000. An example of a code 1000 message was given in the previous section. See Appendix C for a comprehensive list.

5.6 Array and substring errors

There are two errors which may be produced from a program unit which has been compiled with the bound checking option (see chapter 2):

```
++++ ERROR 1050: array bound error
```

An illegal array subscript has been used.

```
++++ ERROR 1051: substring bound error
```

An illegal substring has been used.

5.7 Input/output errors

Input/output errors are those which may be trapped by use of the END= and ERR= specifiers in FORTRAN 77 statements. If these are not used, an error message and code are produced as described below; otherwise execution continues, with the error code available by use of the IOSTAT specifier.

All the messages have the general form:

```
++++ ERROR N: PREFIX UNIT - reason
```

N is the error code; PREFIX describes the IO operation being attempted (it may be OPEN, CLOSE, BACKSPACE, ENDFILE, REWIND, or READ/WRITE), and UNIT is the unit number, with * given for one of the asterisk units and 'internal' for an internal file. The rest of the message gives more information about the error.

End of file on input may be trapped with the END= specifier. The IOSTAT value in this case is -1. If END= is not used, then the message `end of file` is produced, with code 1000. Other errors may be trapped with the ERR= specifier. The IOSTAT value is the corresponding error code, as listed in Appendix C.

5.8 Tracing

Tracing a program's execution is a very useful debugging technique, applicable when a program compiles and runs successfully, but produces unexpected output. Tracing is achieved by including special trace routines in the source code of a program, and is requested by the compiler 'T' option. Tracing output is produced when:

1. entering the program unit;
2. leaving the program unit;
3. a labelled statement is about to be executed;
4. the THEN clause of an IF...THEN or ELSEIF...THEN construct is about to be executed;
5. the ELSE clause of an IF...THEN or ELSEIF...THEN construct is about to be executed;
6. a DO statement is about to be executed; and
7. another subprogram unit is about to be invoked.

In addition three routines; TRACE, BACKTR, and HISTOR, are available for explicit calls by the user. If the main program is compiled with the 'T' option specified, the TRACE point will output a message which starts with '***T' followed by the type of trace point encountered:

For example:

```
*** T entering main program
```

This is followed by a prompt:

```
Tracing enabled?
```

which expects a response of Y or N. A typical trace of the program's execution is:

```
*** T DO at line 3 (1)
```

This line indicates that a DO statement is about to be executed at line 3, and that this is the first call of this statement. For some categories of trace point, a count (modulo 32768) is also given of the number of times this trace point has been met. The routine called TRACE can be called at a particular place within the program. It needs a single LOGICAL argument to turn this tracing information on and off.

For example:

```
CALL TRACE (.TRUE.)
```

turns tracing on. Counting will continue even if the trace output is turned off, so the values produced will be correct if tracing is turned on again later in the program.

HISTOR (short for HISTORY) outputs information about the last few traced subprogram calls. Each line of history information consists of a name, which may be preceded by -> or by <-. A right arrow indicates a traced call of a subprogram, a left arrow indicates a traced exit from a program unit, and a line with neither type of arrow indicates a traced entry to a program unit. Note that the name given when tracing entry and exit from a program unit is the name of the program unit itself, which may not be the name of the entry called by the user. Note that history information is stored only if the subprogram was compiled with the 'T' option.

BACKTR (short for BACKTRACE) outputs information about the current nesting of program unit calls. The routine should be given a single logical argument; if this is .TRUE., then the HISTOR subroutine is invoked after the backtrace information has been produced.

6 FORTRAN 77 with other languages

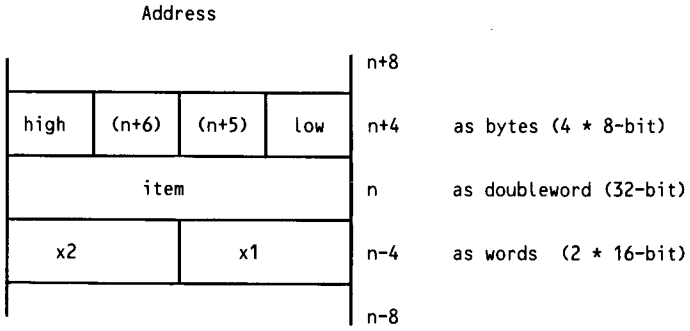
6.1 Introduction

This is a review of the outline code mechanisms of Acorn 32000 FORTRAN 77. It is intended to include sufficient detail to enable (a) a program written in 32000 FORTRAN 77 to call (via an interface procedure) a procedure written in any high-level language which conforms to the Acorn Inter-Language Calling Standard (this includes the Panos interface procedures), or in assembler, and (b) to permit programs in other languages to call 32000 FORTRAN 77 procedures. A working knowledge of the NS32000 CPU architecture and instruction set is useful to explain the low-level technical details.

32000 FORTRAN 77 differs somewhat from most other languages implemented on Acorn 32000-based products. While the specification of parameters and/or results for a 32000 FORTRAN 77 subprogram may be written in other languages, and calls may be made to it, the parameter passing mechanism operated is unconventional. This is primarily a consequence of the definition of FORTRAN 77.

Conventions used in diagrams

Memory is represented diagrammatically in a method such that 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 overleaf.



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

6.2 Parameters

All parameters in 32000 FORTRAN 77 are passed by reference to a store location; the location may be that of a program variable, or of an anonymous value. Thus both variables (which may be updated by the called procedure) and arbitrary expressions (which have a value but which cannot logically be assigned to) may be passed. Hence the following example is valid 32000 FORTARAN 77 and will work:

```
SUBROUTINE JIM (I, J)
   IF (J .GT. 2) I = J + 4
   ...
END

SUBROUTINE FRED
   INTEGER  Q, K1, K2
   ..
   CALL JIM (K1, K2)
   ..
   CALL JIM (Q, 3)
   ..
   CALL JIM (7+K1, 1)
END
```

In the first call on JIM, the addresses of K1 and K2 are passed. In the second, the address of Q and that of a private location containing the value 3 are passed: on this call, Q will be updated. In the third call the value of the expression 7 + K1 is calculated and assigned to a private location, then the address of this and of a further private location (holding the value 1) are passed. All references within JIM to the parameters I and J are made indirectly via the addresses which were passed to it.

The general rule is that for actual parameters which are simple variables or array elements (i.e. which have an address as well as a value), the address of the actual item is passed. In all other cases a private location is allocated by the compiler, which at the time of the call will contain the value of the expression given as the actual parameter. The value is calculated either at compile time or at run time, depending on the nature of the expression: in the case of simple literal values, the constant is stored in the (nominally read-only) code area, and for any other case the value is assigned in the static data area immediately before the call. The address of the appropriate location will be passed to the called procedure. Note that the compiler does not detect at compile time whether an attempt will be made to update a parameter which was not a variable in the actual call (e.g. if the statement CALL JIM (7, 5) was included in the example above, it would be compiled, but the effect of executing it would not be defined).

The parameters to a procedure are not passed directly on the stack, as would be implied by the procedure calling standard, but indirectly. Every call made by or to a 32000 FORTRAN 77 procedure (i.e. a SUBROUTINE or FUNCTION) pushes exactly one item on the stack: this is the address of a parameter vector. The vector comprises a sequence of doublewords (32-bit), which are the addresses of the parameters, as described above. For calls made by 32000 FORTRAN 77 procedures this vector resides in the static data area of the calling module, but there is no requirement that the same should be true when calling an 32000 FORTRAN 77 procedure. Figure 3 overleaf shows the situation immediately after the CXP to/from a 32000 FORTRAN 77 subroutine with N parameters.

For access to parameters, the 32000 FORTRAN 77 compiler generates code which on entry to a procedure saves the old FP on the stack, and copies the vector-address argument to the FP. It is then possible to access the actual parameter locations by references of the form 0(k*4(FP)), i.e. memory-relative addressing via the vector. Immediately before the

procedure returns, the old FP is popped from the stack; control is passed back to the caller by the instruction:

```
RXP    4
```

which also removes the single vector-address argument.

This use of the FP is unconventional, but works well in the context of FORTRAN where all data is normally static, and hence the stack is needed only for control and parameter information storage. It also satisfies the requirements of the inter-language calling standard, in that the FP is preserved across a call.

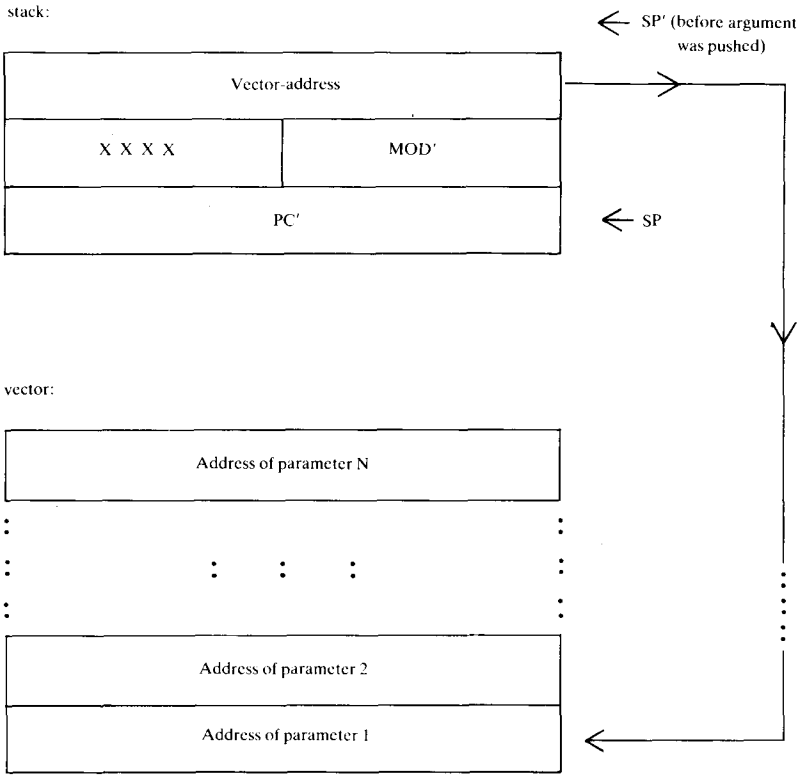


Figure 3

An important point to note is that, as mentioned above, the form in which a FORTRAN parameter list is expressed is different in FORTRAN itself from that in which it must be expressed to conform with the calling standard. In particular all FORTRAN procedures, including those which have no parameters, must be described and referenced from other (more standard) languages as having a single argument: a pointer to a record containing pointers to the parameters proper. The same is true when writing procedures in some other language which are intended to be called from FORTRAN. This obviously limits the scope of inter- language calling with 32000 FORTRAN 77: it will normally be possible to write a specification for, and call, a 32000 FORTRAN 77 procedure from another language, but only procedures which conform in this detail can be called from 32000 FORTRAN 77.

Hence in the latter case, if some arbitrary existing procedure is to be called from a FORTRAN program, it will typically be necessary to write an interface routine which accepts the 32000 FORTRAN 77 call and calls the target procedure in the appropriate manner, converting between parameter access mechanisms as required.

Again it must be stressed, ALL procedures in 32000 FORTRAN 77 are called, and call other procedures, with a single parameter. In the case of a procedure with no arguments, a NIL pointer (vector-address = 0) is pushed as argument to the call, to be removed on return by 'RXP 4'. Hence any interface procedure designed to be called from 32000 FORTRAN 77 MUST be defined with a single argument, even if this will not be used; the consequences of failing to do this are likely to be disastrous.

Care should also be taken in regard to the treatment of values and variables to be passed into or out of a 32000 FORTRAN 77 procedure. As was implied earlier, FORTRAN does not have true 'value' type parameters; the normal technique for passing a value to a 32000 FORTRAN 77 procedure is to have a temporary local variable to which the value is assigned: the address of this variable is then passed in. When writing a procedure to be called from 32000 FORTRAN 77, only parameters which are intended as 'reference' type should be modified by the called procedure; otherwise peculiar effects are possible (as in the example of JIM (7, 5) above). The classic example of this is one in which a supposed constant actually has its value changed during program execution, with potentially baffling results. If

the following program is compiled and run, the number printed is 3 rather than 1! (This only gets through because although the compiler stores the constant 1 in the (read-only) code area, there is no memory protection active to prevent its modification. With memory management hardware in use, the program should fail at run time.)

```
      PROGRAM FUNNY
      CALL FRED (1)
      CALL JIM (1)
      END

      SUBROUTINE FRED (J)
         J = 3
      END

      SUBROUTINE JIM (K)
         WRITE (*, 20) K
20    FORMAT(I5)
      END
```

6.3 Data types

A number of different basic data types are permitted in 32000 FORTRAN 77; these are:

```
INTEGER
LOGICAL
REAL
DOUBLE PRECISION
CHARACTER*K
COMPLEX
ARRAYS
```

Note: the type CHARACTER is identical to the type CHARACTER*1. In addition, arrays formed from one of these basic types may be declared; an array may have from 1 to 7 dimensions. The data storage for these types is as follows:

INTEGER

32-bit (4-byte) signed quantity. The compiler normally aligns all integers on a 4-byte boundary, although the 32000 architecture does not enforce this - it is merely slower to access non-aligned data.

LOGICAL

8-bit (1-byte) unsigned quantity, aligned on 4-byte boundary. The representation of .TRUE. is 1, and .FALSE. is represented as 0. 32000 FORTRAN 77 compiled code tests logical variables by comparing with 0. Note that since the compiler always aligns logical variables on a 4-byte boundary, but only generates byte operations when accessing them, the top three bytes of the storage space allocated may contain undefined data - this is important when passing a logical variable to a non-32000 FORTRAN 77 context: the correct thing to do is to use MOVZBD for this purpose.

REAL

32-bit (4-byte) 32000 standard 'F'-type floating point quantity, aligned by the compiler on a 4-byte boundary for performance reasons.

DOUBLE PRECISION

64-bit (8-byte) 32000 standard 'L'-type floating point quantity, aligned on a 4-byte boundary.

CHARACTER*K

Data of this type is maintained as a contiguous string of K * 8-bit bytes, representing the characters in ASCII. Strings are normally manipulated via string descriptors. A string descriptor is a compound item comprised of 2 doublewords. The first is a pointer to the first character of the string, the second an integer which defines the length of the string (i.e. K). Hence for example the character constant 'Hello' is stored as shown in figure 4.

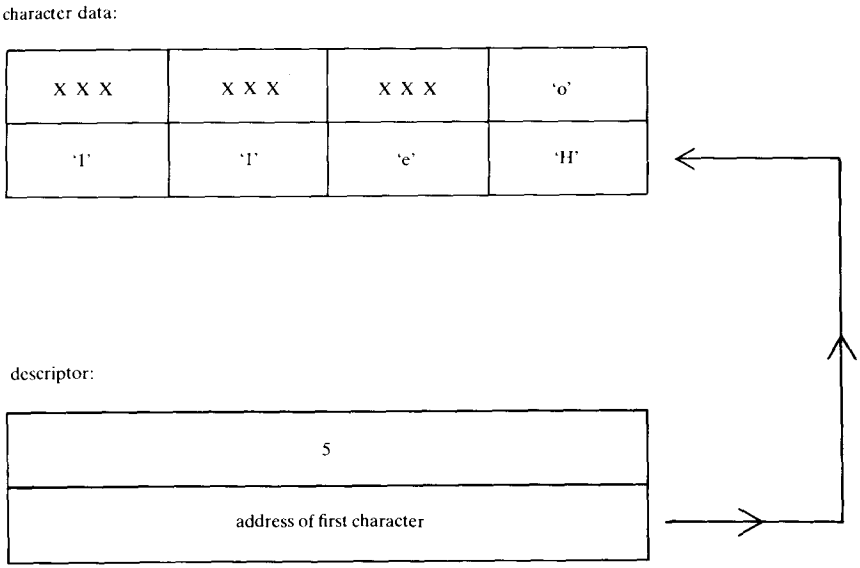


Figure 4

String descriptors are normally located by the compiler in the static data area of the module concerned.

COMPLEX

Complex data consists of two consecutive data items of type REAL, the first representing the real component and the second the imaginary component. Complex variables are aligned by the compiler on a 4-byte boundary.

ARRAYS

Arrays are implemented as contiguous areas of memory containing the elements of the array in the normal 32000 FORTRAN 77-defined order, i.e. the left-most subscript indexes sequentially stored items. Each element of an array will be aligned on the address boundary specified above for the type of the elements, i.e. in general a 4-byte boundary.

6.4 Parameter passing

An actual parameter to a FORTRAN 77 subroutine or function may be:

1. a simple constant of any type except CHARACTER*K
2. a simple variable/array element of any type except CHARACTER*K
3. an expression of any type except CHARACTER*K, excluding cases 1 and 2
4. a character constant
5. a character variable/array element, or a substring thereof
6. a character expression, excluding cases 4,5
7. an array of any type
8. a subroutine or function
9. an alternative return specifier

Of these, all except type 9 require information to be present in the parameter vector for the corresponding item (see below for more details of alternative return parameters). In every case except one, the information is a single 32-bit pointer, but the details of what it points to vary according to the nature of the parameter, as follows:

1. Simple constant

The pointer refers to a storage location of the appropriate size, containing the value. This will be allocated within the code area, which is nominally a read-only area and hence must not be written to.

2. Variable/array element

The pointer refers to the actual storage space allocated for the variable or array element. This will be in either the local static data area for the calling module or a common area, and hence it will be writeable.

3. Non-simple expression

The pointer refers to a private storage location. If the value of the expression is actually constant at compile-time, it will simply be assigned to the location. Otherwise the value of the expression will be computed at run-time, and then assigned. In both cases the assignment takes place immediately before the procedure is called, every time the statement is executed.

4. Character constant

The pointer refers to a string descriptor (as above). The characters of the string will be stored sequentially in the code area, as for other constant (type 1) parameters.

5. Character variable/array element or substring

The pointer refers to a string descriptor whose length field contains the declared length of the variable or array element, or the effective length of the substring. In the latter case if the substring selector expressions are not simple constants then the length will be calculated at run-time immediately before the call takes place.

6. Other character expression

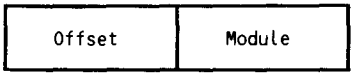
The pointer refers to a string descriptor for the string value which results from evaluating the expression at run-time. The storage space for this value is allocated within the static area of the module performing the call. The actual length of the string value is calculated at run-time as part of the evaluation, and is then assigned to the length field of the descriptor.

7. Array

The pointer refers to the same item that it would if the actual parameter were specified as the first element of the array rather than the array itself, i.e. for all except CHARACTER*K arrays, it refers to the first actual array element, and for CHARACTER*K arrays it refers to a standard string descriptor for the first element.

8. Subroutine or function

This parameter type is the only one in which the item in the parameter vector is not a simple pointer: in this case it is a standard 32000 procedure descriptor, i.e. an object of the form:



which is accessed by the CXPD instruction to transfer control to the procedure.

9. Alternate return parameter

No parameter vector space is occupied: see the later section on subroutines with alternate return parameters.

6.5 Results

The 32000 FORTRAN 77 implementation differs in a further significant respect from the mechanism defined by the inter-language calling standard, in regard to the return of function results. Essentially, all functions return (in R0) not the result value itself, but the address of a location containing it. The data must then be fetched from that address by the calling procedure, after the function has returned. On this basis, all FORTRAN functions should be specified and used from other languages as returning a pointer to data of their result type. The exception to this rule is when a function returns data of type CHARACTER, or CHARACTER*N; in this case the function is implemented with a hidden extra parameter which precedes the normal parameters in the parameter vector. This item is a pointer to a string descriptor (as described earlier). Such a function is implemented as effectively a subroutine with an initial parameter which is of the same type as the function result. The compiler allocates private space for this object, not in the module where the function itself resides, but in each module which calls the function. The size of the space is determined by the specification of the function result size in the calling module. In particular this enables functions whose actual definition specifies result type CHARACTER*(*) to work.

6.6 Subroutines with alternate return parameters

No actual parameter information corresponding to the return labels is passed to such subroutines; instead they are implemented as functions returning an integer value (pointed at by R0 in the normal manner). The value returned is that given by the expresssion following the RETURN in the called subroutine. On return from the subroutine, the calling procedure checks the value of the result, and if it lies in the range 1 to (number of '*label' type actual parameters) inclusive, passes control (by means of a case jump on the value) to the appropriate label; otherwise control is passed to the point after the call statement.

6.7 Modules and Naming

In logical terms, each program unit (SUBROUTINE, FUNCTION, PROGRAM or BLOCK DATA) defines a separate module, and is so implemented in AOF, i.e. the compiler generates a complete independent AOF module for each program unit. The name of the module corresponds to the name of the program unit, except that in the case of un-named BLOCK DATA the module name is 'F_BLDT'. Identifiers in FORTRAN are normally limited to 6 characters; however the compiler does accept longer ones, giving a warning at each. The maximum length of an identifier is 255 characters, which is also the limit in AOF. The compiler generates an AOF name the same as the source name, except that all lower-case letters on input are translated to upper-case.

First Example: Calling a Pascal routine from FORTRAN 77

See the *32000 Pascal Reference Manual* for information on Pascal cross-calling mechanisms.

Suppose a Pascal module Graphics_IO is defined which exports, amongst other items, a procedure Draw_Polygon. This is defined in (extended) Pascal as:

```
CONST MaxVertex = 100;

TYPE GColour = (Black, White, Red, Green, Blue, Yellow);
     Vertex = RECORD vx, vy: Integer END;
     VertexArray = ARRAY [1..MaxVertex] OF Vertex;

EXPORT PROCEDURE Draw_Polygon (colour: GColour; vertex_count: Integer;
                               vertices: VertexArray);
```

It is desired to call this from a FORTRAN 77 program. The procedure is to be callable as if it were specified in FORTRAN 77 as:

```
SUBROUTINE  Draw Polygon (colour, vertex count, vertices)
INTEGER  colour, vertex count, vertices (1:vertex count*2)
```

where the vertex array is mapped as successive pairs of integer values in a FORTRAN integer array.

To convert the call that the FORTRAN 77 program will make into one suitable for the Pascal procedure, an interface procedure must be written.

This will be illustrated in both Pascal and assembler. The first point to note is that because of the limitations of FORTRAN 77 in the syntax of external name specification, it will not be possible to define the procedure using exactly the same name. Hence the name as seen from FORTRAN 77 is to be changed to GIODP (Graphics IO Draw Polygon).

Interface Module in Pascal

```
MODULE Graphics_IO_to_F77_Interface;

CONST MaxVertex = 100;

TYPE
   GColour = (Black, White, Red, Green, Blue, Yellow);
   Vertex = RECORD vx, vy: Integer END;
   VertexArray = ARRAY [1..MaxVertex] OF Vertex;

   F77_GIODP_par_vec =
      RECORD
         colour_p:       †GColour;
         vertex_count_p: †Integer;
         vertices_p:     †VertexArray;
      END;

IMPORT PROCEDURE Draw_Polygon ALIAS 'Graphics_IO~Draw_Polygon'
                    (colour: GColour; vertex_count: Integer;
                     vertices: VertexArray);

EXPORT PROCEDURE  GIODP (VAR v: F77_GIODP_par_vec);

   BEGIN
      Draw_Polygon (v.colour_pt, v.vertex_count_pt, v.vertices_pt)
   END;

END.
```

Notes

As described in the Pascal Reference Manual, it is vital that an interface module such as this be compiled with the checks for assignment of variables turned off, in order to prevent the program failing when the pointer fields of v are tested, since they are not in fact standard Pascal pointer values. The extended Pascal option must be turned on.

The type GColour in Pascal will be implemented as a single byte value. In consequence care should be taken to ensure that only values in the range 0..5 are supplied by the FORTRAN 77 program which calls this interface module. Alternatively the interface procedure could be modified to take an Integer as the colour parameter (i.e. `colour_p: ↑Integer` in the parameter vector record), and convert acceptable values into GColour values, taking appropriate action on unacceptable ones.

Interface Module in Assembler

```
        MODULE  Graphics IO to F77 Interface

        IMPORTC Draw_Polygon = 'Graphics_IO'.'Draw_Polygon'

        EXPORTC GIODP
GIODP
        MOVD    8(SP), R0
        MOVD    8(R0), TOS      ; pass address of parameter 3
        MOVD    4(R0), R1       ; get address of parameter 2
        MOVD    0(R1), TOS      ; pass value of parameter 2
        MOVD    0(R0), R1       ; get address of parameter 1
        MOVD    0(R1), TOS      ; pass value of parameter 1
        CXP     Draw_Polygon
        RXP     4

        END
```

Second Example: Calling a Panos-provided function

Although a library is provided to enable FORTRAN 77 programs to use the facilities of Panos, an example is given here to illustrate the general technique.

Suppose it is desired to make use of the procedure 'SetGlobalString' which is accessible in the Panos module 'GlobalString'. This is defined as:

```
SetGlobalString (STRING:GlobalStringName
                 STRING:GlobalStringValue); INTEGER:Result
```

It seems reasonable to map this into FORTRAN 77 as an INTEGER FUNCTION with the specification:

```
SETGLOBALSTRING (NAME, VALUE)
CHARACTER*(*) NAME, VALUE
```

The necessary interface routine could be implemented either in Pascal or in assembler. The former is illustrated below.

Pascal

In Pascal there is no single representation for the Inter-Language Calling Standard 'String' type parameter. Instead a value parameter of this type is mapped as two 32-bit parameters. For use within Pascal the first of these would normally be specified as (a pointer to) an array of characters and the second as an integer giving the string length. However in this case the Pascal code will perform no manipulation of the strings, so it is simpler to treat both objects as 32-bit integers. Implementing the FORTRAN 77 result mechanism is achieved by assigning the result to a local STATIC variable (i.e. one whose value is held in static rather than stack storage space), and returning as the function result the address of this object. The code is as follows:

```
MODULE F77_Panos_GlobalString_Interface;

TYPE
   F77_string_descriptor =
      RECORD
         Adr: Integer; {rather than CharStringRef}
         Len: Integer
      END;
   F77_string_ref = ↑F77_string_descriptor;

   F77_SETGS_par_vec =
      RECORD
         Name,
         Value: F77_string_ref
      END;

STATIC  SETGS_result: Integer;

IMPORT FUNCTION SetGlobalString ALIAS
                'GlobalString$SetGlobalString'
                   (Name_addr, Name_len,
                    Value_addr, Value_len: Integer): Integer;

EXPORT FUNCTION SETGS (VAR par_vec: F77_SETGS_par_vec): Integer;

   { This procedure is suitable for calling from F77. The }
   { result is defined as an Integer rather than ↑Integer }
   { so that the extended Pascal function ADDRESS (which  }
   { returns an Integer result) may be used to reference  }
   { the static result variable readily (and efficiently).}

   BEGIN
      WITH par_vec DO
         SETGS_result := SetGlobalString (Name↑.Adr, Name↑.Len,
                                          Value↑.Adr, Value↑.Len);
      SETGS := ADDRESS (SETGS_result)
   END;

END.
```

As with the earlier example, it is essential that this interface module be compiled without assignment checks included, by use of the -NOCHECKS or at least the -NOASSIGN option to the Pascal compiler. The consequences of not doing so are not defined.

7 The Interface Library

7.1 Introduction

The FORTRAN 77 compiler generates code which does not conform directly to the Panos Inter-Language Calling Standard, although it is in some respects compatible. As a consequence, it is not in general possible for programs written in FORTRAN 77 to make use of the set of procedures Panos provides. For this reason, a library of AOF modules has been produced, which permits programs written in FORTRAN 77 to call most of the standard procedures defined by Panos. Interfaces are provided to procedures in all of the modules listed in the *Panos Programmer's Reference Manual* with the exception of modules 'Handler' and 'Loader'. Essentially the library defines an interface procedure, designed to be called from FORTRAN 77, for each corresponding Panos procedure.

The library is provided in the file 'ifp-lib' on the FORTRAN 77 distribution disc, and this should be linked in with any FORTRAN 77 program which makes calls to Panos via the library. An example of a link command to do this would be:

```
-> link prog,sub1,sub2  f77,ifp
```

7.2 Naming conventions

Apart from the special cases given below, the name of an interface procedure (i.e. subroutine or function) as seen from FORTRAN 77 is the name of the procedure as given in the Programmer's Reference Manual, with a two-letter prefix 'IF'. The name may be written in any mixture of upper and lower case - identifiers in FORTRAN 77 are not case- sensitive. The exceptions to this rule are the procedures in module 'IO' which take blocks of data as parameters (BlockRead & BlockWrite, with their X- and S- variants). These are defined in two forms, one which is suitable for use with data of type CHARACTER*N (or arrays of CHARACTER), and the other for use with other types of data (INTEGER, REAL etc). For convenience, procedures of the second form are defined under two names

(although the code is the same) for use in the same context with parameters of type REAL or INTEGER, since the FORTRAN 77 compiler insists on consistency of usage of one procedure within another. (The EQUIVALENCE mechanism may be used if block I/O operations on data of other types, LOGICAL, COMPLEX etc., are desired. In these cases the name as seen from FORTRAN 77 has a suffix 'C' for the first form (for CHARACTER*N), and 'I' or 'R' for the second. Note that these names are typically greater than 6 characters in length, and hence the F77 compiler will generate messages on the occurrence of each long identifer. These are purely warnings however (which may be suppressed) and do not themselves cause the compilation to fail.

Examples

```
Panos module  Panos procedure     FORTRAN 77 name

          IO  FindInput           IFFINDINPUT
          IO  ReadByte            IFREADBYTE
          IO  XSBlockWrite        IFXSBLOCKWRITEC (CHARACTER data)
          IO  SBlockRead          IFSBLOCKREADI   (INTEGER data)
          IO  BlockWrite          IFBLOCKWRITER   (REAL data)
  DecodeArg   Substitute          IFSUBSTITUTE
TimeAndDate   BinaryTime          IFBINARYTIME
      Random  XSetRandomSeed      IFXSETRANDOMSEED
```

7.3 Calling conventions

In the Programmer's Reference Manual, Panos procedures are defined in an abstract format with a list of parameters and a list of results, either of which may be empty. Instead of giving a complete list of all the procedures defined in the interface library with the format of a call on each procedure in FORTRAN 77 terms, a set of translation rules is supplied below to convert from the abstract format to the format necessary in FORTRAN 77. Several examples are also given to illustrate this. These rules hold for all cases except certain procedures in module 'IO', which are described in note 4 below.

In general, for a Panos procedure, the format of the corresponding FORTRAN 77 interface procedure is derived as follows:

1. (a) If the Panos procedure has no result(s), the interface procedure will be a SUBROUTINE;

   (b) if the Panos procedure has a first or only result which is of type INTEGER, CARDINAL, HIDDEN or ADDRESS, then the interface procedure will be an INTEGER FUNCTION;

   (c) if the Panos procedure has a first or only result which is of type BOOLEAN, then the interface procedure will be a LOGICAL FUNCTION (and must be declared as such in a type-declaration statement);

   (d) if the Panos procedure has a first or only result which is of type STRING, then the interface procedure will be an INTEGER FUNCTION, whose first parameter must be a variable of type CHARACTER*N, or a substring thereof; it must not be a string constant or other string expression. The string result of the procedure will be assigned to the object which was specified, and the INTEGER value returned by the function will be the number of characters in the object which are actually significant as the result. The object should be of sufficient size to contain the result expected, otherwise the Panos procedure will fail and cause an exception to be signalled;

   (e) if the Panos procedure has a first or only result of type RECORD (of any format) then the interface procedure will be a SUBROUTINE whose first parameter must be an INTEGER ARRAY of one dimension whose size is sufficient to contain all the fields of the record result. On return from the procedure, the record fields will have been written into the elements of this array in the obvious manner.

2. If the Panos procedure has more than one result, then the second and any further results are returned by updating variables supplied as parameters to the interface procedure; these are passed in the same left-to-right order as the results of the procedure are given in the Programmer's Reference Manual, and come BEFORE parameters corresponding to the actual parameters in the abstract procedure definition. The various extra result types are treated as below:

   (a) for a result of type INTEGER, CARDINAL, HIDDEN or ADDRESS, an INTEGER variable must be passed, to receive the returned value;

(b) for a result of type BOOLEAN, a LOGICAL variable must be passed;

(c) for a result of type STRING, two items are passed: the first is a variable of type CHARACTER*N (the comments given in 1(c) in relation to this item apply here also); the next parameter must be a variable of type INTEGER to receive the length of the string result;

(d) for a result of type RECORD, an INTEGER ARRAY must be passed - this is treated in the same way as is defined in 1(d).

3. For each parameter to the Panos procedure, the interface procedure has a corresponding parameter, in matching left-to-right order. The various possible types are handled as below:

(a) for each parameter of type INTEGER, CARDINAL, HIDDEN or (except for the IO block operations, see note 4 below) ADDRESS, a general expression of type INTEGER should be passed;

(b) for each parameter of type BOOLEAN, a general expression of type LOGICAL should be passed;

(c) for each parameter of type STRING, a general expression of type CHARACTER*N should be passed (any value of N is acceptable);

(d) for each parameter of type RECORD (of any format) an INTEGER ARRAY of one dimension and suitable size should be passed;

(e) for each parameter which is a PROCEDURE, a SUBROUTINE or FUNCTION which has the appropriate type and number of parameters should be passed. Note that this typically requires the use of an EXTERNAL declaration for the name of the subroutine or function. To define the format of the user procedure in FORTRAN 77 terms, the same mapping process should be applied to it as to any ordinary Panos interface procedure (see note 3 below).

Notes:

1. In the rules given above, the term 'variable' covers both a simple FORTRAN 77 variable and an array element of the appropriate type.
2. FORTRAN 77 CHARACTER*N FUNCTIONS are not used for string results because:

(a) the FORTRAN 77 rules on assignment of character expressions allow over-long results to be truncated without error, whereas Panos procedures returning strings which do not fit in the supplied buffer will fail, generating an exception; and

(b) in FORTRAN 77, results which do not fill the variable are automatically padded on the right with spaces, so if a Panos procedure returned a string result having trailing spaces, this could not be detected in FORTRAN 77. This may be significant in some contexts.

3. The FORTRAN 77 definition of a user procedure (i.e. a FUNCTION or SUBROUTINE) which is to be passed as a parameter to an interface procedure is derived in the same way as if it were an interface procedure itself, i.e. by application of the rules above to its abstract definition. However it is necessary to observe the following points when writing an actual procedure to be passed in this context:

(a) for a parameter to the user procedure of type STRING, the formal parameter to the procedure should be specified to be of type CHARACTER*(*), i.e. a character string whose length is defined by the calling rather than the called procedure.

(b) any parameter (to the user procedure) which is specified in the abstract definition to be of type STRING or RECORD must not be modified by the procedure, or by any other to which it may be passed, but parameters of any other type may freely be modified.

4. Special case: Block I/O operations

The following procedures from module 'IO' are treated in a manner different from the rules given above:

BlockRead XBlockRead BlockWrite XBlockWrite
SBlockRead XSBlockRead SBlockWrite XSBlockWrite

In each case where the Panos procedure has a parameter of the form 'ADDRESS:Buffer', the interface procedure takes, not an expression of type INTEGER, but a variable or array parameter, such that the data will be read into or written from the FORTRAN 77 data object, rather than a buffer whose address is passed. This is because there is no primitive function in FORTRAN 77 to yield the address of a piece of data, and hence no easy way to perform block I/O on FORTRAN 77 data, if the rules given earlier are applied strictly in this case. As was mentioned in the section 'Naming

conventions', three variants of each of these procedures are provided, with 'C', 'I' and 'R' suffixes. The 'C' variant MUST be used when Block I/O on CHARACTER data is to be performed, since the parameter mechanism for strings is different from that for other types of data. The 'I' and 'R' versions are identical, but are provided for ease of use in maintaining type consistency, as the compiler requires. For CHARACTER data, the effective length of the data object passed is ignored and the 'BLength' parameter instead determines how many bytes of data are read or written. Because of the way arrays are passed in FORTRAN 77 (i.e. by the address of the first element) it is equally possible to read data into a part of an array as into the whole, by specifying as parameter the first element to be written to, rather than just the array name itself. Also note that it is possible to give an arbitrary character expression as the data parameter to the 'C' versions of the BlockWrite procedures.

Examples

1. Panos specification:

```
Module "IO":

SWriteByte (CARDINAL:Stream, CARDINAL:TheByte);
INTEGER:Status
```

FORTRAN 77 use:

```
INTEGER  status, stream, char
...
status = ifswritebyte (stream, char)
```

2. Panos specification:

```
Module "Error":

GetErrorInformation (INTEGER:Error); INTEGER:Result,
                                STRING:Information

XGetErrorInformation (INTEGER:Error); STRING:Information
```

FORTRAN 77 use:

```
INTEGER  result, status, infolen
CHARACTER info*60
```

```
...
status = ...
...
result = ifgeterrorinformation (info, infolen, status)
...
infolen = ifxgeterrorinformation (info, status)
```

3. Panos specification:

```
Module "DecodeArg":

DecodeInit (STRING:KeyString, STRING:ArgumentString);
        INTEGER:Result, HIDDEN:Handle

XDecodeInit (STRING:KeyString, STRING:ArgString);
HIDDEN:Handle
```

FORTRAN 77 use:

```
PARAMETER filekey = 'FILE/a/e-dat'
INTEGER result, handle
CHARACTER  parambuff * paramsize
...
parambuff = ...
...
result = ifdecodeinit (handle, 'COUNT/c/a UPwards/s',
parambuff)
...
handle = ifxdecodeinit (filekey, parambuff)
```

4. Panos specification:

```
Module "Convert":

BooleanToString (BOOLEAN:bool); INTEGER:Result,
STRING:ResultString
```

FORTRAN 77 use:

```
LOGICAL p, q
CHARACTER answer*5, conclusion*40
INTEGER  status, anslen
...
status = ifbooleantostring (answer, anslen, p .or. q)
...
conclusion = 'The statement is ' // answer(1:anslen)
```

5. Panos specification:

Module "Convert":

XStringToBoolean (STRING:sourcestring);
BOOLEAN:BooleanResult

FORTRAN 77 use:

```
LOGICAL decided, ifxstringtoboolean
CHARACTER value*30
INTEGER j, k
...
decided = ifxstringtoboolean (value(j:k))
```

6. Panos specification:

Module "TimeAndDate":

BinaryTime (); INTEGER:Result, RECORD(BTim):BinaryTime

XBinaryTime (); RECORD(BTim):BinaryTime

FORTRAN 77 use:

```
INTEGER result, bintime(0:1)
...
result = ifbinarytime (bintime)
...
CALL ifxbinarytime (bintime)
```

7. Panos specification:

Module "TimeAndDate":

TextualTimeOfBinaryTime (RECORD(BTim):BinaryTime);
INTEGER:Result, STRING:TTime

FORTRAN 77 use:

```
INTEGER  status, bintime(0:1), datelen
CHARACTER thedate*20
...
status = iftextualtimeofbinarytime (thedate, datelen, bintime)
```

8. Panos specification

Module "IO":

BlockRead (CARDINAL:BLength, ADDRESS:Buffer);

INTEGER:Result, CARDINAL:BytesRead

XBlockRead (CARDINAL:BLength, ADDRESS Buffer);
CARDINAL:BytesRead

BlockWrite (CARDINAL:BLength, ADDRESS:Buffer);
INTEGER Result, CARDINAL:BytesWritten

XSBlockWrite (CARDINAL:Stream, CARDINAL:BLength,
ADDRESS:Buffer)

FORTRAN 77 use:

```
INTEGER  status, bytesread, outstream, usednames, terminal
INTEGER  first, last
CHARACTER*9  names(1:16)
INTEGER  numbers(1:16)
REAL  rank(1:32), xdata1
DOUBLE PRECISION xdata(1:35)
EQUIVALENCE (xdata1, xdata(1))
...
status = ifblockreadc (bytesread, 16*9, names)
...
bytesread = ifxblockreadi (16*4, numbers)
...
bytesread = ifxblockreadr (32*4, rank)
...
CALL ifxsblockwritec (outstream, usednames*9, names)
...
CALL ifxsblockwritec (terminal, 15, 'Buffer cleared' // char(10))
...
status = ifblockwritei (byteswritten, (last-first + 1)*4,
+                       numbers(first))
...
CALL ifxblockwriter (35*8, xdata1)
```

9. Panos specification:

Module "File":

Expand (STRING:WildName, PROCEDURE(uproc):ProcessProc,
HIDDEN:ProcessArg, BOOLEAN:TargetIsDir);
INTEGER:Result

uproc : (STRING:Instance, HIDDEN:ProcArg); INTEGER:Status

FORTRAN 77 use:

```
EXTERNAL processfilename
INTEGER  result
...
result = ifexpand ('*-f77', processfilename, 1, .FALSE.)
...

FUNCTION processfilename (filename, arg)
INTEGER processfilename, arg
CHARACTER filename*(*)
...
processfilename = 0
...
END
```

Appendix A

Front end error messages

Note that some of the messages are duplicated because they are caused by slightly different errors.

Class 1

The list below gives all of the class 1 error messages:

Statement not allowed here
FORMAT is not labelled
ENTRY inside DO or block IF
ENTRY not allowed
Brackets not matched
Statement not recognised
Unmatched apostrophe
Statement not recognised
Unknown type
Expecting letter
Expecting letter
'<letter>' already set
expression is not constant
EXTERNAL not allowed in BLOCK DATA
Bad COMMON name
Expecting digit
Integer expected
Expecting name
not array element
'<name>' is not an integer variable
Inconsistent use of '<name>'
Illegal use of '<name>'
Invalid Keyword
UNIT not given
Label '<number>' already used

Illegal structure
Expression is not LOGICAL
Illegal structure
Integer expression required
'<name>' is not an integer variable
Unexpected character '<character>' after GOTO
Illegal expression type
Expression is not LOGICAL
Invalid Keyword
UNIT/FILE not given
UNIT and FILE both given
Invalid Keyword
UNIT not given
RETURN not allowed in main program
THEN must follow 'IF (lexp)'
Illegal DO terminal statement
Assignment to '<name>' not allowed
Invalid Keyword
UNIT not given
Expecting '<name>' here
Statement label expected
Zero is not allowed as a statement label
Statement label too long
Bad format or I/O list
Bad format or I/O list
No Keyword
Invalid Keyword
Bad implied DO variable
Bad type for implied DO variable
FMT=* not allowed here
Bad type for UNIT
END= not allowed
Bad type for UNIT
Bad internal file
Bad type for UNIT
Bad type for UNIT
Bad type for UNIT
Bad type for UNIT
Bad type for UNIT
Bad/missing UNIT expression

'<name>' assumed size
Bad complex constant
length *(*) for function '<name>'
'(' not allowed here
'<name>' not allowed here
Substring not allowed here
Expecting name
COMPLEX relations?
Unknown operator
Bad complex constant
expecting name or constant
Bad Hollerith constant
Bad Hollerith constant
Illegal hex constant
Bad character constant
Empty character constant
Bad logical constant?
Illegal statement in logical IF
Statement not allowed in BLOCK DATA
Expecting end of statement
Expecting '<character>'
Item too long

Class 2

The list below gives all class 2 error messages:

illegal DO terminal statement
ENTRY in FUNCTION has alternate returns
More than <number> lines in statement
Continuation marker not allowed here
Bad end of file
FUNCTION may not have alternate returns
'<name>' already used
Inconsistent use of '<name>'
Unexpected ','
Substring expressions not constant
'<name>' is not an intrinsic function
Inconsistent use of '<name>'
Blank common not allowed here

Bad lower bound expression for '<name>'
Bad upper bound expression for '<name>'
Incompatible declaration for '<name>'
'<name>' in bad equivalence
/<name>/ in SAVE but not COMMON
Common block /<name>/ partly CHARACTER
'<name>' in bad equivalence
EQUIVALENCE involving '<name>' partly CHARACTER
'<name>' not allowed in SAVE
Integer constant expression required
'<name>' in substring in EQUIVALENCE
'<name>' wrong number of bounds in EQUIVALENCE
'<name>' not allowed here
'<name>*(*)' not allowed
Subscripts not constant
Substring expressions not constant
'<name>' not allowed here
'<name>' not in named COMMON
'<name>' is not local
Integer constant required here
Constant required here
Integer expression required
Integer expression required
Integer expression required
Illegal expression in implied DO
Keyword already used
Illegal type for DO variable
Variable required
Keyword already used
Keyword already used
Keyword already used
UNIT not integer
Illegal structure
Illegal jump to label '<number>'
Keyword already used
Wrong type
Not variable or array element
Wrong type
Bad format identifier
Bad I/O list item for READ

'<name>' has assumed size
Label '<number>' is undefined
Unclosed DO or block-IF
Operands for concatenation not CHARACTER
subscript not integer
'<name>' wrong number of subscripts
substring lower bound not integer
substring upper bound not integer
Bad arguments for '<name>'
Wrong number of arguments for '<name>'
Type mismatch,arg <number>
Illegal type conversion
Illegal type conversion
Illegal operand types
Illegal type for operand
'<name>' not allowed here
'<name>' not COMMON
Bound for '<name>' not constant
'<name>*(*)' not argument
Concatenation includes *(*) item
'<name>' not allowed here
Odd number of hex digits
Odd number of hex digits
Bad exponent
Illegal type after '-'
'<name>' cannot be a dummy argument
'<name>' occurs more than once
'<name>' is not CHARACTER
'<name>' is CHARACTER
Illegal character '<character>' in label
Zero statement label not allowed
Illegal jump at line <number>
Unclosed DO/IF block
Statement label already used
Inconsistent use of '<name>'
Label '<name>' already used in different context
Statement labelled '<number>' is non-executable
Illegal reference to label '<number>'
Inconsistent use of '<name>'
Inconsistent use of '<name>'
Type of '<name>' already set

Appendix B

Code Generator error messages

argument out of range for CHAR
: The intrinsic function CHAR has been used with a constant argument outside the range 0-255.

local data area too large
: The size of the local storage area for the program unit exceeds 2,147,483,647 bytes.

array <name> has invalid size
: The size of the given array is negative or exceeds 2,147,483,647 bytes.

attempt to extend common block <name> backwards
: An attempt has been made to extend a COMMON block backwards by means of EQUIVALENCE statements

bad length for CHARACTER value
: A value which is not positive has been used for a CHARACTER length.

<class> storage block containing <name> is too large
: <class> is local or COMMON. The storage block containing the named variable exceeds 2,147,483,647 bytes.

concatenation too long
: The result of a CHARACTER concatenation may exceed 2,147,483,647 characters.

conversion to integer failed
: A REAL or DOUBLE PRECISION value is too large for conversion to an integer.

D to R real conversion failed
: A DOUBLE PRECISION value is too large for conversion to a REAL.

DATA statement too complicated
: The variable list in a DATA statement is too complicated. It must be simplified.

division by zero attempted in constant expression
The divisor might be REAL, INTEGER, DOUBLE PRECISION or COMPLEX.

real constant too large
A REAL constant exceeds the permitted range.

double constant too large
A DOUBLE PRECISION constant exceeds the permitted range.

inconsistent equivalencing involving <name>
The given variable is involved in inconsistent EQUIVALENCE statements.

increment in DATA implied DO-loop is zero
A DATA statement implied DO loop has a zero increment.

insufficient store for code generation
The code generator has run out of workspace. The program unit being compiled must be simplified.

insufficient values in DATA constant list
There are more variables than constants in a DATA statement.

integer invalid for length or size
A value which is not positive has been used for a CHARACTER length or array size.

lower bound exceeds upper bound in substring
In a substring, a constant lower bound exceeds the constant upper bound.

lower bound of substring is less than one
A constant substring lower bound is less than one.

upper bound exceeds length in substring
A constant substring upper bound exceeds the length of the character variable.

stack overflow - program must be simplified
The internal expression stack has overflowed. The offending statement must be simplified.

subscript below lower bound in dimension N
a constant array subscript is less than the lower bound in the given dimension.

subscript exceeds upper bound in dimension N
: A constant array subscript exceeds the upper bound in the given dimension.

too many constants in DATA statement
: There are more constants than variables in the DATA statement.

type mismatch in DATA statement
: The type of the constant is illegal for the corresponding variable.

variable initialised more than once in DATA
: A variable has been initialised more than once by DATA statements in this program unit.

wrong number of hex bytes for constant of TYPE type
: A hex constant has been given with the wrong number of digits.

zero increment in DO-loop
: A DO loop with a constant zero increment value has been used.

inconsistent use of <name>
: The external subroutine or function <name> has been used with inconsistent argument types.

The previous error message would occur with the following program:

```
call abc(1.0)
call abc(2)
end
```

Appendix C

Run-time error messages

Code 1000 errors

bad operands for double precision **

d1**d2 where d1 is negative

bad operands for real **

r1**r2 when r1 is negative

operand too large in DASIN

abs(arg) in DASIN or DACOS exceeds 1

operand too large in ASIN

abs(arg) in ASIN or ACOS exceeds 1

<ch> edit descriptor cannot handle logical list item

Format descriptor used with a LOGICAL list item is not L; <ch> is the actual descriptor used.

invalid logical in input

Formatted input file D contains bad logical value.

<ch> edit descriptor cannot handle character list item

Format descriptor used with a CHARACTER list item is not A; <ch> is the actual descriptor used.

<ch> edit descriptor cannot handle numeric list item

Invalid descriptor for numeric value. <ch> is the actual descriptor used.

Z field width unsuitable

Wrong number of digits in hex (Z) input field for given type.

invalid number in input

Bad number (range or syntax) in formatted I, D, E, F or G input.

FORMAT - unexpected character <ch>

Invalid character <ch> in FORMAT.

FORMAT - bad numeric descriptor

Bad syntax for numeric FORMAT descriptor.

FORMAT - cannot use ' when reading

Quoted string used in input FORMAT.

FORMAT - unexpected format end

End of FORMAT inside quoted string

FORMAT - cannot use H when reading

nH used in input FORMAT.

FORMAT - bad scale factor

Bad +nP or -nP construct.

FORMAT - too many opening parentheses

More than 20 nested opening parentheses (including the first).

FORMAT - trouble with reversion

No value has been read or written by the repeated part of the format (this would cause an infinite loop if not trapped).

The following program fragment illustrates the 'trouble with reversion' format error:

```
      write(1, 10) i, j
   10 format(i5, (1x))
```

FORMAT - width missing or zero

Bad width in numeric edit descriptor

Bad complex data

Bad COMPLEX constant in list directed input.

LD repeat not integer

Repeat count (r*) in list directed input is not valid

LD input data not REAL

Syntax or range error in REAL list directed input value.

LD input data not INTEGER

Syntax or range error in INTEGER list directed input value.

LD input data not DP

Syntax or range error in DOUBLE PRECISION list directed input value.

LD input data not LOGICAL

Syntax error in LOGICAL list directed input value.

LD input data not COMPLEX

Syntax or range error in COMPLEX list directed input value.

LD input data not CHARACTER

Syntax error in CHARACTER list directed input value

LD repeat split CHARACTER

Attempt to split a repeated character constant across a record boundary. This is strictly legal, but almost impossible to implement correctly.

Unformatted output too long

Unformatted record length exceeds maximum permitted. This can occur with direct access output only.

Unformatted input record too short

Input record does not contain sufficient data.

mismatched use of ACCESS, RECL in OPEN

ACCESS = 'DIRECT' has been quoted in an OPEN which does not contain a RECL specifier, or vice versa.

Input/output errors

invalid unit number

Unit number not in range 1-60.

invalid attribute

Invalid attribute used in OPEN statement.

duplicate use of file name

The same file name has been used more than once in an OPEN statement.

invalid unit for operation

BACKSPACE/REWIND/ENDFILE attempted on unit connected for direct access.

error detected previously

An IO error has been detected previously on this unit, and trapped with ERR=.

direct access without OPEN

A direct access READ or WRITE has been used without an OPEN statement for the unit.

invalid use of unit

Inconsistent use of unit (formatted mixed with unformatted, sequential mixed with direct access or ENDFILE done previously).

input and output mixed

Input and output mixed on a sequential unit (without intervening REWIND or OPEN).

direct access not open for input

The direct access file could not be opened for input (e.g. file is write only).

direct access not open for output

The direct access file could not be opened for output (e.g. file is read only).

end of file on output

An attempt has been made to write off the end of a sequential file (in practice, this will occur with internal files only).

not available

BACKSPACE operation is not available.

bad unformatted record (message)

A record in an unformatted file does not have the required structure.

invalid access to terminal file (message)

Attempt to use terminal (or other output device) as an unformatted or direct access file. More detail is given

sequential open failed (message)

The actual reason for the failure (e.g. 'Bad name') is given in the brackets.

direct access open failed (message)

The actual reason for the failure (e.g. 'Bad name') is given in the brackets.

direct access IO failed (message)

For example, attempt to read past end of file.

record length too large

The record length specified in a formatted direct access OPEN exceeds the permitted maximum (512 bytes).

bad direct access file (message)

A file used for direct access has invalid initial data or insufficient record length.

sequential write failed (message)

I/O error on sequential output (e.g. Can't extend)

Index

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200