FORTRAN 77

reference manual

ARM Evaluation System

Acorn OEM Products

FORTRAN 77

Part No 0448,013
Issue No 1.0
15 August 1986

ISBN 1 85250 008

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

Contents

1. Introduction

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 FORTRAN 77. This manual describes the use of Acorn FORTRAN 77 running under ARM Executive ; note that it is not a tutorial.

The Acorn 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. Less technical approaches are provided in *A Structured Approach to FORTRAN 77 Programming* by T M R Ellis, published by Addison Wesley, and *A Pocket Guide to FORTRAN 77* by Clive Page, published by Pitman.

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 ARM computers.

If you want to use the floating point emulator, refer to the *ARM UTILITIES reference manual.*

1.1 Installation

FORTRAN 77 for the ARM is distributed on ADFS format floppy discs.

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. This program is in Acorn object format (AOF) and is linked into an executable form using a linker program. A single command file is normally used to run both parts of the compiler and the linker. 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 executes a command file which runs the two parts of the compiler and the linker in sequence, and so compiles the program without the need for the user to give three 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.

A help file called README on the Fortran distribution floppy disc, contains details about the file structure needed for the f77 command to work as described in this manual.

You can write your own command files (see *ARM UTILITIES reference guide*) for running the Fortran compiler in a different way, using other file structures. If you do this, it is advisable to use a different command name and leave f77 consistent with the examples in this manual.

There is also a command f77q, which compiles more quickly because it does not produce program listings and maps. The map gives the name, type and location of local and common variables in each program unit.

To run the two parts of the compiler and the linker separately, the command f77fe is used for the front end, f77cg is used to generate the machine code, and link is used to link AOF files into relocatable image files (rif).

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.

The form of the command line is as follows:

```
f77 {-source} name { {-to name} {-list name}
    {-error name} {-opt options} {-map name}
    {-help} {-m} {-link name} {-library name}
    {-image name} }
```

where name stands for a user-supplied filename, and options represents a list of one or more compiler options. Braces enclose optional items. The arguments can be given in any order. Explanations of each follow:

-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. The file must be in directory $.f77_files.source.

-list {name}
: Unless the quick version of the compiler command file, f77q, is used, a listing of the compiled program along with line numbers of the source is generated. This listing can be sent to a file or device specified in the argument (for example, printer:). If no filename or device is specified, then the compiler sends the listing to a file whose name is based on the source filename, for example, a file called Fprog in the source directory will have a listing file called $.f77_files.list.Fprog.

-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.

-to name
: The Acorn Object Format output file of the compiler is given a name based upon the source filename by default that is, a file called fort1' in the source directory will be given the AOF name $.f77_files.AOF.fort1'. The -to argument can be used to specify an alternative name.

-map name

Unless the quick version of the compiler command file, f77q, is used, a storage map of the compilation is produced by the code generator and sent to the filename specified. This will be put into the directory `$.f77_files.map`.

-help

The help argument displays a reminder of all the other arguments you can use with the f77 command. The reminder is a list of keywords as used at the start of command files (see *ARM UTILITIES reference guide*); /a indicates that an argument is compulsory, /k that it is optional. For explanations of the use of the arguments, refer to the manual (this section), as none is given in response to use of the help argument.

-m n The module limit option is used to define the limit on the number of program units (main program and subprograms) in a compilation. A limit is necessary because of the chunk file structure of object files. The default limit is 10. To set the limit to 15, use the option -m 15.

-link name

If you use the -to argument to rename the AOF file, then the link argument must be used to direct the linker to take its input from the AOF file.

library name

This argument is used to change from the default run-time library, and give the compiler the name of the file to use instead (see *ARM UTILITIES reference manual*).

-image name

The image file contains the final executable output from the linker. The image argument is used to send the output to a file with a different name from the default one.

You cannot change the name of the file for the output from the front end of the compiler. It always has the name of the source file and is in directory `$.f77_files.fcode`.

2.1.1 Example compiler commands

The minimal command

```
f77 Fprog
```

This command compiles the source program Fprog, with default directories. The compiler expects the source code to be in file $.f77_files.source.Fprog'. The output will be directed to the following files:

the fcode is in `$.f77_files.fcode.Fprog`
the AOF code is in `$.f77_files.AOF.Fprog`
the program listing is in `$.f77_files.list.Fprog`
the map is in `$.f77_files.map.Fprog`
the image code is in `$.f77_files.image.Fprog`

Error messages are directed to the VDU, and default compilation options are used (see section 2.2).

If the quick compiling command:

```
f77q Fprog
```

is used, the result is the same except that no list or map file is produced.

Specifying the input and output files

```
f77 -source Fprog -to $.Afiles.Fprog -list $.f77_files.list.proglist
 -link $.Afiles.Fprog -error printer
```

The compiler front end expects the source code to be in file`$.f77_files.source.Fprog`. The AOF code is in `$.Afiles.Fprog`, so the linker has to be redirected to find its input in `$.Afiles.Fprog`. The rest of the output is directed to the default files as in the minimal command, except for the program listing, which is in `$.f77_files.list.proglist`.

C. Using some options and specifying a map file:

```
f77 Fprog -opt +tW0 -map $.map.FProg
```

The program Fprog' is compiled, with tracing specified in the code, warning messages inhibited, and a storage map sent to the file $.f77_files.map.Fprog'.

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, 6 and 7 are enabled or disabled by preceding them with + or -. The options X, L, M 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 (for example, 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
>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 (for example, 100/106). The error will lie in this range.

M n

This sets the module limit. (10)

T

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

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 (for example, 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.

6

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

7

This option is used to control warnings about the use of FORTRAN 77 language extensions. If unset, warnings are not produced; otherwise messages are produced if the warning level (Wn) is 2 (the default) or greater. The option is unset by default, so that the extensions may be used without messages, whatever the warning level.

2.3 Compiling in separate stages

As an alternative to using the f77 command to execute a command file that runs both parts of the compiler, the two parts can be run separately. This section gives details of how to run the front end and the code generator separately.

2.3.1 Front end

The FORTRAN 77 front end accepts the full language as defined in the ANSI standard, together with a number of extensions. Upper- and lower-case letters are equivalent in all contexts, and are always converted to upper case in the FCODE output (for example, ::x=abc(y):: and ::X=ABC(y):: would both refer to the function ::ABC::). Identifiers may contain up to 255 characters.

Command format

The front end is a BCPL command with the following argument string:

```
"FROM/A,TO=FCODE/K,LIST/K,OPT/K,VER/K"
```

FROM	FORTRAN 77 source program.
TO	FCODE output file. If this argument is not quoted, no FCODE is produced.
LIST	Listing file. If LIST is quoted, a listing of the source program with line numbers is sent to the file, together with any error messages. Otherwise error messages are sent to the initial output stream, and no listing is produced.
OPT	Front end option string. The options available are described below.
VER	Output file for compiler messages and errors; if omitted output is to the terminal.

Options

+	turn following options on
-	turn following options off
T	include tracing
Xn	width of cross-reference output (no output if width is 0).
Wn	Set warning level
7	Strict FORTRAN 77 mode
6	FORTRAN 66 mode

The default settings are X0W2-T67.

The +T switch causes the front end to embed calls to special trace routines at various points in the program, such as program unit entry, DO statements, labelled executable statements and subprogram calls. See the later section on tracing for more details.

The +W option controls the production of warning messages: level 0 suppresses them, level 1 permits the most significant, and 2, 3 and 4 allow gradually less serious warnings to be produced.

The +X option controls the width in which cross-reference information is written, with a width of zero causing the output to be supressed. 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 referred to. 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 referred to.

The FORTRAN 66 switch (+6) is intended to help in the compilation of programs written in the previous version of FORTRAN. When set, most constructs which have different meanings in the two versions are interpreted according to the FORTRAN 66 definition. In particular:

(a) DO loops will always execute at least once

(b) Hollerith (nH) constants are allowed in DATA and CALL statements, and quoted constants in calls are not of CHARACTER type.

(c) Non-CHARACTER array names are allowed as format specifiers.

The strict FORTRAN 77 option (+7) is used to control warnings about language extensions. If unset, warnings are not produced; otherwise messages are produced if the warning level (Wn) is 2 (the default) or greater. The +7 option is unset by default, so that the extensions may beused without messages, whatever the warning level.

When the FORTRAN 66 switch is used, Hollerith and quoted constants are treated in the same way when used as arguments in CALLs - 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')
call jim(4habcd)
```

The +6 option must also be used with the code generator if it is wished touse Hollerith constants in DATA statements.

If the FORTRAN 66 switch is set, run-time FORMATs specifiers may also be non-CHARACTER array names.

For example:

```
double precision d(3),num
data d(1),d(3)/8h(1X,D20., 5h,I5/)/
data num /2h10/
...
d(2) = num
...
write(6, d) 2.3d0, 10
...
```

This facility was introduced to assist in the implementation of FORTRAN 66 programs; it is strongly recommended that new programs use CHARACTER formats.

Examples

```
f77fe f77.prog -to tmp.fcode
```

Compile source program in f77.prog to FCODE in tmp.fcode.

```
f77fe f77.prog -ver x
```

Compile f77.prog, producing no FCODE output, with messages sent to the file x.

```
f77fe f77.prog -to tmp.fcode -list list.prog
```

Compile as before, but also send source listing to the file prog in directory list.

```
f77fe f77.prog -to tmp.fcode -opt t
```

Compile with tracing calls included.

2.3.2 Code generator

The code generator takes an FCODE file and produces an object file and/or assembler output. It requires a BCPL system with at least 800 globals, and access to the floating point instruction set for certain operations involving REAL and DOUBLE PRECISION constants.

Command line

The code generator is a BCPL command with the following argument string:

```
"FCODE/A,OBJ=TO/K,ASM=A/K,VER/K,OPT/K,MAP/K"
```

FCODE	FCODE input file.
TO	Object output file. If this argument is not quoted, no object file is produced. The object file is in an AOF file, and may be merged with other AOF files using the linker Link to produce a large compiled program (see section 2.4).
ASM	Assembler output file. If this argument is quoted, a disassembled version of the object code is sent to the file.
VER	Output file for code generator messages and errors; if omitted output is to the terminal.
OPT	Option string. The options available are described below.
MAP	The file used for the code generator map output. The MAP gives the name, type and location of local and COMMON variables in each program unit. The location is relative to the start of the static area for a local variable and is the offset in the block for a COMMON variable. The offset of each statement number from the start of the code is also given.

Options

+	turn following options on
-	turn following options off
B	insert code for array and substring bound checking
6 or H	allow Hollerith constants in DATA statements
Ln	set line number level.
Mn	set module limit (10).

The default options are L1-B6. The front end options T, 7, W and X are ignored by the code generator, whilst the front end ignores B and L, so that the same option string may be given to both programs, if required (except for the M option – this should be specified separately).

The line number option (+L) is used to control the amount of line number information included in the code. The possible levels are:

0	No line numbers.
1	Lines containing subprogram calls are recorded.
2	Calls and lines which could cause an exception are recorded.
>=3	Every line is recorded.

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

The module limit option (+M) is used to define the limit on the number of program units (main program and subprograms) in a compilation. A limit is necessary because of the chunk file structure of object files. The default limit is 10.

Examples

```
f77cg tmp.fcode -to obj.prog
```

Code generate from FCODE in tmp.fcode to an object file in obj.prog.

```
f77cg tmp.fcode -asm vdu:
```

Code generate tmp.fcode, sending assembler output to the terminal.

```
f77cg tmp.fcode -to o.prog -map map.prog
```

Code generate as before, but also send map output to map.prog.

```
f77cg tmp.fcode -to obj.prog -opt +bm30
```

Code generate with bound checking code inserted and the module limit increased to 30.

2.4 Linking and execution

A compiled FORTRAN program is linked using the standard ARM linker. The FORTRAN 77 library file should be quoted as one of the input files, using the library qualifier /l. The resulting program is run in the normal way. For details of how the linker works, see *ARM UTILITIES reference manual.*

An example of a link command is:

```
link obj.prog,obj.sub1,$.xlib.f77/l -image prog -adfs
```

3. Extensions to the standard

Acorn FORTRAN 77 offers several enhancements to the standard which are described in this chapter. Further extensions concerning input/output are described in chapter 4. To get a warning that these extensions have been used, use the 7 option when compiling (see chapter 2).

3.1 Hexadecimal constants

Acorn 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 (that is 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 EXAMPLE*(*)
PARAMETER (EXAMPLE = ?H1C05141E0C)
J = ?I1234
```

Here, example' 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/

..

D(2)=NUM

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 Acorn FORTRAN 77, all lower- case letters (except in FORMATs and character constants) are converted into upper case upon reading the source. Thus all statements, identifiers and so on may be in lower case. Names up to 255 characters long may be used. It is worth noting, to save confusion, that there is no limit on the length of CHARACTER values.

```
DO v = v1,v2,v3      or      DO WHILE (logical expr)
...                          ...
...                          ...
END DO                       END DO
```

END DO may not be used as the terminal statement in a labelled DO loop.

This form of loop is compatible with VAX/VMS FORTRAN 77.

3.5 Random number generators

Acorn FORTRAN 77 has two routines for random number generation:

```
REAL FUNCTION RND01()
```

returns a pseudo-random number in the range 0.0 <= r < 1.0

```
SUBROUTINE SETRND (I)
```

selects a new random sequence; if I is zero the sequence is non-repeatable. The generator is initialised with a call to SETRND(0) so that successive runs will produce different sequences.

3.6 Include statement

An include statement allows a file containing source code to be read in by the compiler at the point where the include statement occurs. The syntax for the statement is:

```
INCLUDE filename'
```

Line numbers in the include file are not recorded on the object file and will therefore not appear in a backtrace; correct line numbers are shown in the program listing and error messages.

4. Input/output

This chapter describes how Acorn 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. Unit numbers in the range 1 to 60 may be used, as well as the two * units for the keyboard and screen. Note that your filing system limits the number of files you can open simultaneously. Consult your filing system manual.

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*}

that is 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
```

This associates the file FILE with unit 10.

```
PROG DATA 32=DATA 3=X
```

This associates DATA with unit 1, data with unit 32, and x with unit 3.

The two * units always refer to the screen and the keyboard. Any units which are not connected to a file in an OPEN statement or command line

3.4 Loops

3.4.1 WHILE ... ENDWHILE

This loop construct has the syntax:

```
  WHILE (logical expr) DO
  ...
  ...
  ENDWHILE
```

WHILE and ENDWHILE must be nested correctly, and neither statement may be used as the terminal statement of a DO-loop, or in a logical IF.

The loop is equivalent to:

```
11IF (.NOT. logical expr) GOTO 12
  ...
  ...
  GOTO 11
12
```

This form of loop is compatible with WATFIV and the Salford FTN77 compiler for PR1ME computers.

3.4.2 DO WHILE

This loop construct has the syntax:

```
  DO n[,] WHILE (logical expr)
  ...
  ...
n ...
```

The rules regarding nesting and the terminal statement are exactly as for normal DO loops.

This form of loop is compatible with the Fujitsu and VAX/VMS FORTRAN 77 compilers.

3.4.3 Block DO

The syntax of DO and DO WHILE loops has been extended so that the terminal statement number may be omitted. The loop is then terminated by an END DO statement.

assignment also refer to these streams.

A file accessed with STATUS=SCRATCH (OPEN) or STATUS=DELETE (CLOSE) is deleted when the unit is closed.

All files are closed automatically when a program terminates.

When writing to a sequential formatted file, a distinction is made between file 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). ESCAPE 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 (0D) and line feed (0A) are each recognised as record terminators. Form feed (0C) 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 (that is 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).

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; however, it would be simple to provide a utility program to create a new direct access file of a given size and record length.

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 (for example, 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 - that is,, 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 (for example, 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. 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.

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 (for example, 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. The strict FORTRAN 77 option (++7++) is used to control warnings about language extensions. If unset, warnings are not produced; otherwise messages are produced if the warning level (Wn) is 2 (the default) or greater. The ++7++ option is unset by default, so that the extensions may be used without messages, whatever the warning level.

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 A 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 A 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	128 Kbytes
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 static data area 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 address of the data area is given in hexadecimal. 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: operands are zero in ATAN2

routine          data area    line

F77_ZSTP         &00005744
F77_ATA2         &00005720
DEF              &00005714     22
ABC              &0000571C     37
F77_MAIN         &00005710      4
```

In this example, the main program (with default name) has called ABC, which has called DEF, which has called ATAN2 (the name shown is the internal name for the intrinsic function ATAN2). The final routine is the main error handler.

The call to ABC in the main program was on line 4; the call to DEF in ABC was on line 37, and so on. The appearance of line numbers in the backtrace is controlled by the compiler L option; level 1 is the default. See chapter 2 for details about compilation options.

If a hardware trap occurs in a program compiled with line number level 1, it may not be possible to determine the exact line number. This is illustrated by the following trace:

```
++++ ERROR 3000: hardware trap

routine          data area    line

ABC              &00005514      5/16
F77_MAIN         &000054EC      3
```

Here, the main program called ABC, which failed with a hardware trap between the lines 5 and 16 inclusive. If the program is recompiled with line level 2, the exact line number will be displayed.

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 B 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 B.

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. The user selects the T option when compiling (see chapter 2) to specify that calls to special trace routines are to be included in the code.

These routines will cause trace information to be output 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

(7) another subprogram unit is about to be invoked

The trace routines will output a message which starts with ***T and indicates the type of trace point encountered; for some of these it will also indicate a count (modulo 32768) of the number of times this trace point has been met. A special routine called ++TRACE++ can be called with a single LOGICAL argument to turn this tracing information on and off. Note that even if the trace output is off, the counting will still be done so the values produced will be correct if tracing is turned on again.

If the main program is compiled with tracing on, the user will be asked if trace output is to be produced or suppressed. If the main program is compiled without tracing, then trace output is initially enabled.

In addition to the ++TRACE++ routine, two further subroutines are supplied as part of the tracing package. The first of these is ++HISTOR++ (short for |HISTORY|), which causes information to be output 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 rather than the name of the entry called by the user.

The final routine provided is ++BACKTR++ (short for lBACKTRACEl) which output information on 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.

In the ARM kernel system, all tracing output is sent to the terminal.

6. Appendix A

6.1 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.

`too many program units in compilation`

The module limit must be increased.

`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
```

7. Appendix B

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.

7.1 Input/output errors

`invalid unit number`

Unit number not in range 1-60.

`invalid attribute`

Invalid attribute used in OPEN statement.

`duplicate use of filename`

The same filename 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 (for example, file is write only).

`direct access not open for output`

The direct access file could not be opened for output (for example, 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 (for example, 'Bad name') is given in the brackets.

`direct access open failed (message)`

The actual reason for the failure (for example, '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 (for example, Can't extend)

Acorn

The choice of experience.