BBC Microcomputer 32016 Second Processor

Acorn FORTRAN 77 user guide

Issue D

The Acorn FORTRAN 77 compiler and related software was developed for the BBC 32016 Second Processor by Topexpress Ltd, Cambridge, England.

All enquiries, comments and suggestions should be addressed to Acorn Computers Ltd as described in the 32016 Second Processor User Guide.

Contents

1. Introduction to Acorn FORTRAN 77

This manual describes the use of FORTRAN 77 running under the Panos operating system on the BBC 32016 Second Processor for the BBC Microcomputer. The language is provided on an Acorn DFS format disc, and before use it must be transferred onto a disc of the appropriate type (ie Winchester for ADFS, the fileserver for NFS, or another floppy disc for DFS).

1.1 The FORTRAN 77 language

FORTRAN has long been established as the programming language most widely used in scientific and numeric applications. FORTRAN 77 is a standard version of the language, and this standard definition has been used in the production of Acorn FORTRAN 77. There are several standard libraries or packages written in FORTRAN, for example the NAG library and the GINO-F graphics system.

1.2 Acorn FORTRAN 77

The version of FORTRAN 77 which has been implemented on the BBC 32016 Second Processor conforms to the ANSI standard for the language (see section 2.1). In addition, useful facilities such as the ability to call routines written in other languages (such as C and Pascal) have been included. The FORTRAN 77 compiler can also recognise many FORTRAN 66 programs, thus providing compatibility with the earlier FORTRAN IV standard.

Like the other compiled languages provided with the 32016 Second Processor, FORTRAN 77 executes under the Acorn Panos operating system. Programs are prepared using the Panos editor, and linked (possibly with modules from 'external' languages) using the Panos linker.

1.3 Installation

On the FORTRAN 77 distribution disc there is a command file called install. The steps involved in installing FORTRAN 77 are:

- Enter the Panos system by inserting the Panos distribution disc and pressing SHIFT BREAK with both the 32016 Second Processor and the host machine powered up.
- Insert the FORTRAN 77 distribution disc into the top drive (drive 0/2) and enter the Panos command

  ```
  install f77 onto :fs
  ```

  where :fs is the desired filing system. If this is ADFS, the Winchester drive should be ready to receive data; if it is NFS you should be logged

to to a fileserver so that files may be transferred; if it is DFS you should have a formatted 80-track disc in the lower drive (drive 1/3).

- The install program will copy the appropriate files onto the filing system specified, creating directories where necessary on the ADFS and NFS. After install has finished, a FORTRAN 77 system will be ready for use in the appropriate filing system. The distribution disc should be put aside for safety.

1.4 Using Acorn FORTRAN 77

To produce an executable FORTRAN 77 program, three main steps have to be carried out. First, the source program is created using a text editor, e.g. the Arcs editor.

The next step is the compilation itself. This is achieved by issuing a command of the form

```
f77 prog
```

which causes the file prog to be compiled into an Acorn object format (AOF) file suitable for linking. The AOF file is automatically called prog-aof. The full format of the f77 command is described later.

The last step of preparing the executable object file is linkage. As with most other compiled languages, the AOF file produced by the compiler is linked with a suitable library or libraries.

The last three steps described above are illustrated with a specific example in the next section.

1.5 Example FORTRAN 77 programming session

In this section, the compilation and linking of small FORTRAN 77 program is described. It is assumed that the source program is in a file called prg which contains the following text:

```
      PROGRAM AVERAGE
      SUM = 0
      DO 10, I=1,10
         READ *,VAL
         SUM = SUM + VAL
   10 CONTINUE
      PRINT *,'AVERAGE IS',SUM/10.0
      STOP
      END
```

The program reads ten number from the keyboard and prints their average on the VDU. It is assumed that the reader is familiar with FORTRAN 77's input/output requirements, and prepares the source file accordingly.

Once FORTRAN 77 is installed as described in section 1.3, compilation is

achieved using the following Panos command:

```
f77 prg
```

This will cause the object code to be stored in the file prg-aof. The compilation is actually achieved in two phases. First the source is converted into an intermediate format which is stored in a file called fcode. Secondly, this is converted into the final AOF file. These two steps are carried out automatically.

Before the program may be executed, it must be linked. This is performed using the command:

```
link prg f77
```

The final relocatable image file (rif) is created by the linker and is called prg-rif. To execute the program, its name is typed as a command:

```
prg
```

Typing in ten numbers with RETURN after each onewill cause their average to be printed out.

2. Description of Acorn FORTRAN 77

This chapter describes Acorn FORTRAN 77 with respect to the differences between it and the standard cited in the next section. These differences are in the form of additions and enhancements to the standard, and limitations.

2.1 Language reference

The published reference for FORTRAN 77 is the ANSI (American National Standards Institute) document ANSI X3.9-1978, entitled 'Programming language FORTRAN'. The document has the ISO (International Standards Organisation) reference ISO 1539-1980 (E).

It is assumed that the reader has access to the document mentioned above, or to a text book based around the published FORTRAN 77 standard.

2.2 Restrictions and variations on ISO 1539-1980

The only restrictions and variations present in Acorn FORTRAN 77 are related to input and output (IO). They are described in detail in chapter 3.

2.3 Extensions to ISO 1539-1980

Acorn FORTRAN 77 provides several enhancements to the standard which are described in this chapter. None of the changes is major, but for maximum portability it is recommended that only standard FORTRAN 77 is used.

2.3.1 Hexadecimal

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

?Tdigits

T 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 (ie. 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 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.

2.3.2 The % character

Acorn FORTRAN 77 allows the use of the character % identifiers. It may be used wherever a letter A to Z might be used. In order to make the compiler recognise the % character, the +% option must be given in the compiler's command line. This is described in detail in chapter 5.

2.3.3 FORTRAN 66 option

Another option (or 'switch') which may be specified in the command line is determines whether the compiler will be in 'FORTRAN 66 mode'. When this is the case, the action of FORTRAN changes as follows:

1 DO loops will always execute at least once

2 Hollerith (nH) constants are allowed in DATA and CALL statements, and quoted constants are NOT of CHARACTER type.

When the FORTRAN 66 switch is used, both Hollerith and quoted constants 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)
```

The code generator H option must be set if it is wished to use Hollerith constants in DATA statements (see chapter 5).

Run-time FORMATs must always be of CHARACTER type, even if the FORTRAN 66 switch is set.

3. Input/Output

This chapter describes how FORTRAN 77 input and output functions are implemented on the 32016 Second Processor and how this affects programs.

3.1. 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.

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, 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:

```
x.prog abc def
```

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

```
x.prog 10=a.file
```

This associates the file 'a.file' with unit 10.

```
x.prog f.data 32=y.data 3=x.y
```

This associates 'f.data' with unit 1, 'y.data' with unit 32, and 'x.y' with unit 3.

The special filename * always refers to the terminal, ie the keyboard and the screen. In addition any units which are not connected to a file in an OPEN statement or command line assignment refer to the terminal.

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:

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

2 Quoting FORM='PRINTER' in an OPEN statement for the unit causes 'printer' output to be assumed for that unit (note that 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.

3.2. Sequential files

3.2.1. Opens and closes3

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
```

3.2.2. 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 Panos BlockRead call, so that the normal BBC Micro line-editing conventions apply. 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 (:ØD) and line feed (:ØA) are each recognised as record terminators. Form feed (:ØC) 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:

character	effect
space	LF
Ø	LF/LF (extra blank line)
1	CR/FF (newpage)
+	CR (overprint)
*	none

The initial LF (space/Ø) 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 automaticlly 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

VDU DRIVERS IN FORTRAN

OPEN RAWVDU: for output.

OPEN (UNIT = 10, FILE = 'RAWVDU:', FORM = 'PRINTER')

3.3. 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.

Direct access file starts with four special bytes which identify it give the record length. These bytes are the characters 'DA' followed by the record length as a two-byte value (LS byte first). It is permissable 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 direct access OPEN is 512 bytes.

If the file has been opened for updating or input, the first four 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 to both 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.

Example program which uses direct access to write and read a file on unit 42:

```
      open(42, access='direct', file='dafile', recl=16
     +     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.

3.4. OPEN and CLOSE

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

3.5. INQUIRE

3.5.1. 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 actualy 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.

3.5.2. 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 (eg. 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 EXIST reply is .true. and the responses to the SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED specifiers are 'UNKNOWN'.

3.6. BACKSPACE

BACKSPACE is allowed only at the start of a file, or immediately after an ENDFILE operation on the unit. In the latter case, the 'end of file' status is cancelled.

3.7. 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. This status may be cancelled by a subsequent BACKSPACE statement.

3.8. 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.

3.9. FORMAT decoding

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

3.9.1. Lower case letters

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

3.9.2. Extraneous repeat counts

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

3.9.3. Edit descriptor separators

Commas can 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).

3.9.4. 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
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 supressed.

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 (eg. 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.

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

3.9.6. Abbreviations and synonyms

0P can be abbreviated to P,
1X to X, and
T1, TL1 and TR1 to T, TL and TR respectively.
A0 is a synonym for A.

3.9.7. 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.

4. Runtime library

Linking of FORTRAN AOF files with the library is performed using the Panos linker as described in section 5.3. The FORTRAN library contains the routines used for such tasks as input and output.

5. Preparing FORTRAN 77 programs

This chapter describes in detail the procedure required for converting a FORTRAN 77 source format into an executable relocatable image file. The steps necessary have already been described, but the various options available (such as FORTRAN 66 mode) are explained here.

5.1 Preparing FORTRAN 77 source

It is assumed that source programs are prepared using one of the system's editors, and that the user knows how to use the editor sufficiently well to do this.

Lines of source are equivalent to images on a punched card system, and it is important that certain fields start in particular columns:

A C or * in column 1 indicates a comment line

Line numbers appear in columns 2 to 5

A non-null, non-space character in column 6 marks a continuation card

The statement proper appears in columns 7 to 72.

The auto-indent capability of the editors makes it easier to keep statements aligned with, say, column 7 (providing that the first column is aligned properly).

5.2 How to compile FORTRAN 77 programs

The Panos command to compile FORTRAN 77 programs has been given already in its simplest form:

```
f77 prg
```

There are, however, several other parts which may appear on the command line. Its full specification is:

```
f77 source [<list> listf] [<opt> opts] [<fcode> codef] [<identify>][<help>]
```

where things between square brackets [...] are optional, words between angles brackets <...> are keywords which must be supplied if the parameter following is given. The meanings of th keywords are:

list — This should be followed by the name of a file to which the compiler will send a line-numbered source listing, in addition to any errors encountered during the compilation.

opt — following this keyword is a list of options to control the action of the compiler. The options are described in the next section.

fcode The compiler takes ~~to~~ phases to convert the source program to an aof file. The first stage, called the 'front end' converts the FORTRAN 77 program into an intermediate form known as FCODE. The second phase (code generation) converts the fcode file to an aof file. The fcode option enables the user to give the name of the intermediate file, which defaults to fcode.

identify If this keyword is cited in the command line, the compiler will identify itself when called. This is useful for checking which version of the compiler is in use.

help The compiler will respond to this keyword by printing help about itself, eg the format of the command line expected.

Examples are:

```
f77 prg
```

Simply compile the source and put the final AOF file in prg-aof.

```
f77 myprog list mylist opt +lt-x identify
```

Compile myprog, produce a compiled listing, set some options, and identify the compiler's version.

5.2.1 Compilation options

There are two sets of compiler options, one for the front end and one for the code generator. These are described separately, front end first.

The format of the option string is a list of characters, each of which may be preceded by a + or - and followed by a parameter. The plus sign enables the option and the minus sign disables it. The option letters are:

NT END:

L This produces a listing if enabled. The listing will be sent to the standard output if no list file was cited, otherwise it will sent to the listing file.

T If this option is enabled, calls to special trace routines are embedded in the object code. Such calls are made on program unit entry, DO statements, labelled executable statements, and subprogram calls. See chapter 8 for details on tracing.

Wn This sets the warning message level. The value of n should be a digit between Ø and 4. Ø suppreses all warning messages. Level 1 allows serious warnings to be printed, level 2 allows slighty less serious warnings through, and so on up to level four which permits all warning to be printed.

Xn This sets the width of the cross-reference output produced by the compiler. X is follwed by a number n which is the width used. Cross reference output begins when the END statement is encountered in the program. It consists of a list of names in the program with the lines at which they were encountered. Also listed are the program labels with the line number at which the label occurred, the type of statement (executable or not) on the line, and the lines which refer to the label.

If the width is cited as Ø, no cross-reference lsiting is produced.

& If this option is enabled, the character & may be used as a letter in identifiers. If the character appears at the start of the name, it is converted into the pair F_ The facility is provided for use when writing library subprograms (modules) for which it is necessary to invent names not normally available.

6 This switch, when enabled, enables the front end to recognise FORTRAN 66-type programs, as explained in section 2.3.3.

The default settings for the front end switches are W2XØ-&LT6, ie warning messages of level 1 and 2 are printed, no cross-reference listing is produced, & it disallowed, no listing is produced (unless list is given in the command line), no tracing code is generated and FORTRAN 66 mode is off.

There are two options available for the code generator. They are:

CODE GENERATOR:

B If this is enabled, the compiler will generate code for subscript bounds checking in array and substring accesses. If an error is detected, the run-time system will give an appropriate message.

H When this option is enabled, Hollerith constants are allowed to appear in DATA statements.

The default settings for the code generator's options are -BH, so to enable either of the options they have to be cited explicitly, eg

```
f77 myprg opt +bt
```

to generate code for bounds checking and run-time tracing. Note that in general the Panos system does not distinguish between upper and lower case, so either may be used in command lines.

5.3 How to link a compiled program

Once a source program has been compiled without error, the AOF file will be created and may be linked using the Acorn linker. The technique for doing this is the same for all of the compiled languages provided with the 32Ø16 Second Processor. The only part that varies is the name of the library used. For FORTRAN 77 this is f77-lib.

To link a single AOF file, the command

```
link prog lib f77
```

would be used. The file prog-rif would be created as a result and could be executed from Panos by typing it as a command. It is also possible to link several AOF files and libraries. See the instructions on how to use the Acorn linker in the Panos User Guide for more details.

6. Running FORTRAN 77 programs

As mentioned above a linked FORTRAN 77 object program may be executed by typing its name from the the Panos command prompt. The name of the program must be accessible through the cli$path variable. For example, the usual directory for RIFs on the DFS is 'r'. If

```
cli$path = "DFS::2."
ext$-rif = "r."
```

then typing the RIF's name at command level (eg fortprg) will be translated into :2.r.fortprg on the DFS.

6.1 Temporary files

The front end produces intermediate code in the fcode file (called FCODE by default). This is used by the code generator. The file has the Panos suffix -tmp so will be put into the directory specified by the ext$-tmp Panos variable, eg "junk.temp.-".

7. Additional facilities

The additional facilities provided by Acorn FORTRAN 77 have already been covered.

8. Errors and debugging

This chapter lists the error messages which are produced by the FORTRAN 77 system at various stages of the compilation and execution of programs.

8.1 Compile-time messages

While the program is being compiled, errors may be detected by the front end and the code-generator. As usual with compiler systems there is a range of error messages, from helpful 'warnings' that the programmer may be bending a rule of FORTRAN 77 systax, to fatal 'system' errors, which cause the compiler to cease operation.

8.1.1 Front end error messages

This section describes errors of class 1 and 2. Class 1 errors are serious errors which cause the front end to abandon the current statement. The statement is printed, together with a line indicating where the error was detected, and a line giving the error number and an explanation. Thus if line 211 contained the faulty FORTRAN statement:

```
101    silly
```

then the message produced might be:

```
211  101    silly
 L  211-------?
Error (code 173): Statement not recognised
```

Each error is identified by a positive number, and for each error number there is an associated descriptive error message. Note that the error messages are not all different; the error number allows experts to know precisely where in the compiler the error was detected, but for normal use the error message should suffice.

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

```
0110  Statement not allowed here
0120  FORMAT is not labelled
0130  ENTRY inside DO or block IF
0141  ENTRY not allowed
0170  Brackets not matched
0171  Statement not recognised
0172  Unmatched apostrophe
0173  Statement not recognised
2000  Unknown type
2001  Expecting letter
2002  Expecting letter
2003  '(letter)' already set
2011  expression is not constant
3020  EXTERNAL not allowed in BLOCK DATA
```

```
3041  Bad COMMON name
3060  Expecting digit
3061  Integer expected
4000  Expecting name
4020  not array element
5000  '(name)' is not an integer variable
5010  Inconsistent use of '(name)'
5011  Illegal use of '(name)'
5020  Invalid Keyword
5022  UNIT not given
5030  Label '(number)' already used
5040  Illegal structure
5041  Expression is not LOGICAL
5050  Illegal structure
5060  Integer expression required
5061  '(name)' is not an integer variable
5062  Unexpected character '(character)' after GOTO
5070  Illegal expression type
5071  Expression is not LOGICAL
5080  Invalid Keyword
5082  UNIT/FILE not given
5083  UNIT and FILE both given
5090  Invalid Keyword
5092  UNIT not given
5100  RETURN not allowed in main program
5110  THEN must follow 'IF (lexp)'
5111  Illegal DO terminal statement
5120  Assignment to '(name)' not allowed
5130  Invalid Keyword
5132  UNIT not given
5150  Expecting '(name)' here
5170  Statement label expected
5171  Zero is not allowed as a statement label
5172  Statement label too long
5190  Bad format or I/O list
5191  Bad format or I/O list
5192  No Keyword
5193  Invalid Keyword
5242  Bad implied DO variable
5243  Bad type for implied DO variable
5260  FMT=* not allowed here
5261  Bad type for UNIT
5261  END= not allowed
5262  Bad type for UNIT
5263  Bad internal file
5264  Bad type for UNIT
5265  Bad type for UNIT
5266  Bad type for UNIT
5267  Bad type for UNIT
5268  Bad type for UNIT
5269  Bad/missing UNIT expression
5270  '(name)' assumed size
7010  Bad complex constant
7011  length *(*) for function '(name)'
7012  '(' not allowed here
7013  '(name)' not allowed here
7030  Substring not allowed here
7060  Expecting name
```

```
7071  COMPLEX relations?
7180  Unknown operator
7190  Bad complex constant
7191  expecting name or constant
7192  Bad Hollerith constant
7193  Bad Hollerith constant
7194  Illegal hex constant
7200  Bad character constant
7201  Empty character constant
7210  Bad logical constant?
8020  Illegal statement in logical IF
8021  Statement not allowed in BLOCK DATA
8080  Expecting end of statement
8090  Expecting '(character)'
8100  Item too long
```

Class 2 errors differ from class 1 errors in that the compiler does not abandon processing of the current statement. Note that in some circumstances, this can lead to spurious errors in the statement being reported.

Note that some class 2 error messages are given not at the line that caused the error but at the point at which the error was detected - thus information on missing labels is given at the end of the program unit (rather than where the label is used), and an attempt to SAVE something that cannot be SAVEd will be reported at the end of the declarations for the program unit.

```
0100  illegal DO terminal statement
0140  ENTRY in FUNCTION has alternate returns
0150  More than (number) lines in statement
0151  Continuation marker not allowed here
0160  Bad end of file
1000  FUNCTION may not have alternate returns
2010  '(name)' already used
3000  Inconsistent use of '(name)'
3001  Unexpected ','
3010  Substring expressions not constant
3030  '(name)' is not an intrinsic function
3031  Inconsistent use of '(name)'
3040  Blank common not allowed here
3050  Bad lower bound expression for '(name)'
3051  Bad upper bound expression for '(name)'
3052  Incompatible declaration for '(name)'
3070  '(name)' in bad equivalence
3071  /(name)/ in SAVE but not COMMON
3072  Common block /(name)/ partly CHARACTER
3073  '(name)' in bad equivalence
3074  EQUIVALENCE involving '(name)' partly CHARACTER
3075  '(name)' not allowed in SAVE
3080  Integer constant expression required
3090  '(name)' in substring in EQUIVALENCE
3091  '(name)' wrong number of bounds in EQUIVALENCE
4001  '(name)' not allowed here
4002  '(name)*(*)' not allowed
4010  Subscripts not constant
4011  Substring expressions not constant
```

```
4012  '(name)' not allowed here
4013  '(name)' not in named COMMON
4014  '(name)' is not local
4015  Integer constant required here
4016  Constant required here
4021  Integer expression required
4022  Integer expression required
4023  Integer expression required
4030  Illegal expression in implied DO
5021  Keyword already used
5031  Illegal type for DO variable
5032  Variable required
5081  Keyword already used
5091  Keyword already used
5131  Keyword already used
5133  UNIT not integer
5160  Illegal structure
5180  Illegal jump to label '(number)'
5194  Keyword already used
5200  Wrong type
5201  Not variable or array element
5210  Wrong type
5220  Bad format identifier
5240  Bad I/O list item for READ
5250  '(name)' has assumed size
6000  Label '(number)' is undefined
6001  Unclosed DO or block-IF
7000  Operands for concatenation not CHARACTER
7020  subscript not integer
7021  '(name)' wrong number of subscripts
7031  substring lower bound not integer
7032  substring upper bound not integer
7040  Bad arguments for '(name)'
7041  Wrong number of arguments for '(name)'
7042  Type mismatch,arg (number)
7050  Illegal type conversion
7070  Illegal type conversion
7072  Illegal operand types
7080  Illegal type for operand
7090  '(name)' not allowed here
7110  '(name)' not COMMON
7120  Bound for '(name)' not constant
7122  '(name)*(*)' not argument
7140  Concatenation includes *(*) item
7141  '(name)' not allowed here
7195  Odd number of hex digits
7197  Odd number of hex digits
7220  Bad exponent
7230  Illegal type after '-'
8000  '(name)' cannot be a dummy argument
8001  '(name)' occurs more than once
8010  '(name)' is not CHARACTER
8011  '(name)' is CHARACTER
8022  Illegal character '(character)' in label
8023  Zero statement label not allowed
8024  Illegal jump at line (number)
8025  Unclosed DO/IF block
8026  Statement label already used
```

```
8030  Inconsistent use of '(name)'
8040  Label '(name)' already used in different context
8041  Statement labeled '(number)' is non-executable
8042  Illegal reference to label '(number)'
8050  Inconsistent use of '(name)'
8060  Inconsistent use of '(name)'
8070  Type of '(name)' already set
```

8.1.2 Warning messages

By use of the W option, the user can instruct the front end to give advice in the form of warnings. The messages are graded on their 'severity'. 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 is used to flag the use of extensions to standard FORTRAN 77, that might give trouble when moving software to another machine. Levels 3 and 4 are used to indicate items that are legal but bad style, and thus possibly mistakes. An example of the last is the use of implicit typing of identifiers, which would be an error (perhaps caused by a misspelling) if it was thought that all identifiers had been explicitly typed.

The warnings are listed below:

```
Level 1:
  0152  last statement not END
  0153  Blank statement - treated as comment
  5140  PRINT treated as WRITE
  5141  WRITE treated as PRINT
  8027  Statement cannot be executed
```

Warning 0153 (blank statement) is produced only when the FORTRAN 66 option is set.

```
Level 2:
  1001  Program name omitted
  7061  Long name '(name)'
  7196  Non-standard hexadecimal constant

Level 3:
  8071  '(name)' typed implicitly
```

8.1.3 Fatal compile errors

The front end performs a number of consistency checks, and will produce a System Error message and abandon the compilation if one of these fails. These errors should not occur in practice: if one does, the user should send a copy of the program to the supplier of the compiler, to enable the fault to be isolated and fixed.

8.1.4 Errors detected by the code generator

Certain compile-time errors cannot be detected by the front end, but are reported instead by the code generator. An example of this is the program:

```
common a,b
equivalence (a,b)
end
```

This would produce the error message:

```
++++ Error near source line 3:
     inconsistent equivalencing involving B
```

The error messages that can be produced by the code generator are:

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 last error would occur with the following program:

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

8.1.5 Code generator limits

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

code size	0128K 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.

8.2 Run time errors

All errors detected by the FORTRAN run-time library produce a message of the following form:

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

N is an error number and 'text' is a description of the error. This message is followed by a backtrace, in which each line 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. 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 link offset may be compared with the statement number map produced by the code generator to find the approximate position of the call. 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:

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

Routine	MOD	data area	link
F_INIT	608	7356	255
F_SQRT	592	7292	41
DEF	544	7232	29
ABC	528	7172	16
F_MAIN	512	7168	18

In this example, the main program (with default name) has called ABC, which called DEF, which 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 at byte offset 18 from the start of the code; the call in ABC to DEF was at offset 16, etc.

8.2.1 Code 1000 errors

There is a series of simple errors which all have code 1000; the text message gives all the necessary information.

These errors are:

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 DEXP

bad operand in DLOG

operand too large in CABS

operand too large in DASIN
    abs(arg) in DASIN or DACOS exceeds 1

operands are zero in DATAN2

operand too large in ASIN
    abs(arg) in ASIN or ACOS exceeds 1

operands are zero in ATAN2

operand too large in EXP

bad operand in ALOG

operand negative in DSQRT

operand negative in SQRT

(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 field 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 - missing opening parenthesis

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 - B must be followed by N or Z

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).

FORMAT - missing closing parenthesis

FORMAT - width missing or zero
Bad width in numeric edit descriptor

FORMAT - number too large

Bad complex data
Bad COMPLEX constant in list directed input.

LD repeat not integer
Repeat count (r*) inlist 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 (255 bytes).

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.

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

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

8.2.2 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:

++++ ERROR 1050: array error near line N

An illegal array subscript has been used. The line number refers to the FORTRAN source file.

++++ ERROR 1051: substring error near line N

An illegal substring has been used.

8.2.3 Input/output errors

Input/output errors are those which may be trapped by use of the END= and ERR= specifiers in FORTRAN statements. If these are not used, an error mesage 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, the following 'reason' is produced, with code 1000:

```
end of file
```

Other errors may be trapped with the ERR= specifier. The IOSTAT value is the corresponding error code, as listed below.

The following codes and 'reasons' may be produced in IO error messages:

1001 invalid unit number
Unit number not in range 1-60.

1002 invalid attribute
Invalid attribute used in OPEN statement.

1003 duplicate use of file name
The same file name has been used more than once in an OPEN statement.

1004 invalid unit for operation
BACKSPACE/REWIND/ENDFILE attempted on unit connected for direct access.

1005 error detected previously
An IO error has been detected previously on this unit.

1006 direct access without OPEN
A direct access READ or WRITE has been used without an OPEN statement for the unit.

1007 invalid use of unit
Inconsistent use of unit (formatted mixed with unformatted, sequential mixed with direct access or ENDFILE done previously).

1008 input and output mixed
Input and output mixed on a sequential unit (without intervening REWIND or OPEN).

1009 direct acccess not open for input

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

1010 direct acccess not open for output
The direct access file could not be opened for output (eg. file is read only).

1011 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).

2000 not available
BACKSPACE operation is not available.

2001 bad unformatted record
A record in an unformatted file does not have the required structure.

2002 invalid access to terminal file
Attempt to use terminal as an unformatted or direct access file.

2003 sequential open failed (message)
The actual reason for the failure (eg. 'Bad name') is given in the brackets.

2004 direct access open failed (message)
The actual reason for the failure (eg. 'Bad name') is given in the brackets.

2005 direct access IO failed
For example, attempt to read past end of file.

2006 record length too large
The record length specified in a direct access OPEN exceeds the permitted maximum (512 bytes).

2007 bad direct access file
A file used for direct access has invalid initial data or insufficient record length.

2009 bad command line syntax

The following message indicates an error during program initialisation, and may not be trapped. However it will will never occur in the Panos system:

open of asterisk units failed

8.3 Tracing FORTRAN 77 programs

When compiling a program unit, the user use the Front End option T to specify that calls to special trace routines are to be included in the code.

These routines will be invoked 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.

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 BACKTRACE) 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.

9. Using FORTRAN 77 with other languages

It is possible to link FORTRAN modules with others written in languages such as Pascal (in fact any other language which produces output conforming to the Acorn Object Format and Acorn Cross-language Calling Standard). In addition, FORTRAN 77 programs may be linked with the standard Panos library routines documented in the Panos Programmer's reference manual.