Desktop Fortran 77

for Acorn RISC OS-based Computer Systems

User Guide

Intelligent Interfaces Ltd

October 1999

All correspondence should be addressed to:-

Technical Queries
Intelligent Interfaces Ltd
P O Box 80
Eastleigh
Hampshire
SO53 2YX

Tel: 023 8026 1514
Fax: 087 0052 1281

E-mail: support@intint.demon.co.uk
URL: http://www.intint.demon.co.uk

CONTENTS

Introduction

For the seriously scientific user the Desktop Fortran 77 package enables an Acorn RISC OS-based computer to be used as a cost effective workstation for developing large Fortran programs.

The !Fortran77 application enables Fortran programs to be compiled, linked and run in the RISC OS Desktop environment. When used with the editor supplied (!SrcEdit) it can 'throwback' errors by highlighting the line containing the error in the source text.

The compiler fully conforms to the ANSI FORTRAN X3.9-1978 standard and. in addition, provides a number of optional extensions.

The package contains the !Fortran77 application, the compiler front end (f77fe), the compiler code generator (f77cg), a choice of linkers (oldlink and newlink), the source editor (!SrcEdit), the symbolic debugger (asd), the IFExt and IFLib utility libraries, which include routines to return the addresses of variables, make SWI calls and read and write memory, and the DrawF, Graphics, SpriteOp, Utils and Wimp public domain libraries. A text file (helpF77) is supplied to enable !SrcEdit to provide on-line help.

The User Guide describes the installation and use of the compiler on Acorn RISC OS-based computers but is not a tutorial on Fortran programming.

The package requires a computer fitted with 4 Mbyte of RAM, a hard disc, RISC OS version 3.1 to 3.71 or RISC OS version 4.02 or greater and is StrongARM compatible.

Conventions Used

Text entered by the user and as it appears on the screen is shown as follows

`This is text as it appears on the screen`

Arguments to commands and options are shown as follows

`-debug` *`arguments`*

The chosen value must be entered for *`arguments`*.

Optional arguments are shown in square brackets

`[-map` *`file`*`]`

Installation

1. Open a directory viewer on a suitable directory for the `FORTRAN` directory on the destination filing system. If this is anything other than the root directory !Fortran77 must be reconfigured as described in the next section.
2. Open a directory viewer on Distribution Disc 1.
3. Drag the `FORTRAN` directory from Distribution Disc 1 to the suitable directory on the destination filing system.
4. Remove Distribution Disc 1 and keep it in a safe place.
5. Open a directory viewer on Distribution Disc 2.
6. Drag the `FORTRAN` directory from Distribution Disc 2 to the suitable directory on the destination filing system.
7. Remove Distribution Disc 2 and keep it in a safe place.
8. Open a directory viewer a suitable directory for the Library Directory on the destination filing system. If this is anything other than the root directory !Fortran77 must be reconfigured as described in the next section.
9. Open a directory viewer on Distribution Disc 3.
10. Drag the `Library` directory from Distribution Disc 3 to the root directory of the destination filing system.
11. On computers running RISC OS 3.60 or earlier, update the !`System` application by dragging the !`System` application from Distribution Disc 3 to the !`System` application of the computer.
12. Remove Distribution Disc 3 and keep it in a safe place.
13. Re-set the computer.

Checking the Installation

1. Open a directory viewer on the `FORTRAN.Examples.General` directory.
2. Double click on the `Test` obey file.
3. The following should be displayed

```
Topexpress FORTRAN 77 front end version 1.19
Program    WORLD Compiled

Total workspace used 6016
ARM FORTRAN 77 code generator version 1.62
Main program (WORLD): code 104; data 20
Total code size: 104; data size: 20
ARM Linker: (Warning) Attribute conflict within AREA F77$$Data
    (conflict first found with rts(F77$$Data)).
ARM Linker: (attribute difference = {0 INIT}).
ARM Linker: (Warning) Symbol Image$$DataLimit referenced,
Image$$RW$$Limit used.
ARM Linker: finished,  1 informational, 2 warning and 0 error
messages.
 Hello Fortran world

STOP

Press SPACE or click mouse to continue
```

Directories

A directory for FORTRAN programs must, in turn, contain the following directories

`f77` contains FORTRAN source text files
`aof` contains Acorn Object Format files for subsequent linking
`aif` contains executable Acorn Image Format files

o contains Acorn Object Format files for subsequent linking when the newer version of the linker is used (see Appendix G).

Configuring the !Fortran77 Application

The `!Fortran77.!Run` obey file sets the following operating system variables to configure !Fortran77:

`F77cl$Dir` points to the directory containing the compiler front end (f77fe), code generator (f77cg) and linker (link), as supplied `<Fortran77$Dir>.^.^.Library`

`F77libs$Dir` points to the directory containing the Fortran libraries, as supplied `!Fortran77.lib`

`MaxF77$Libs` sets the maximum number of libraries, as supplied 20

`MaxF77$Files` sets the maximum number of source or object files, as supplied default 20

The `!Fortran77.!Run` obey file also sets the following operating system variables to configure command line operation options:

`F77$Tmp` points to the directory used for temporary scratch files created during compilation, as supplied `!Fortran77.tmp`

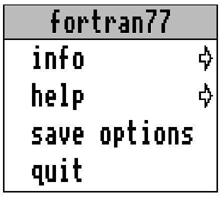
`F77$Lib` points to the directory containing the Fortran libraries, as supplied `!Fortran77.lib`

`Run$Path` points to the directory containing the f77, f77lk, linkf77, d77, df77lk and dlinkf77 commands, as supplied `!Fortran77.Execlib.NewLink.`

Using the !Fortran77 Application

Before carrying out the selected operations, the !Fortran77 application sets the current directory to the directory containing the `f77` directory and creates any `aof` and `aif` directories which it needs and which do not already exist. If any `INCLUDE` files are to be used, they should be stored in this directory so that they do not need a directory prefix.

1 Install !SrcEdit and !Fortran77 on the icon bar. Click 'menu' over the !Fortran77 icon to enable the options to be saved

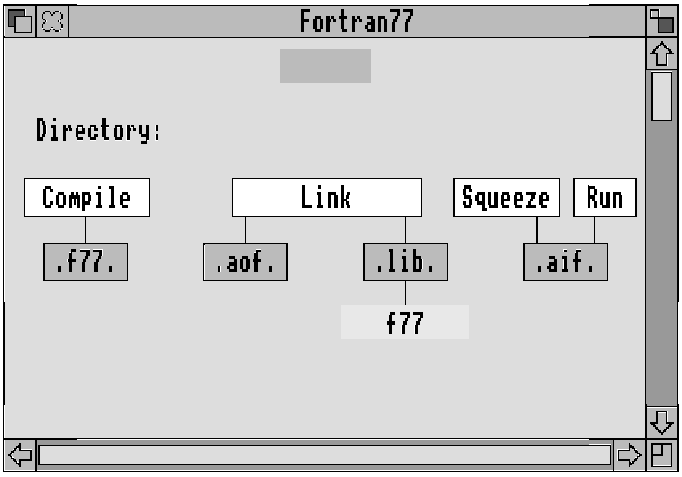


2 To open the main window either

a) drag a source text file from an `f77` directory onto the !Fortran77 icon

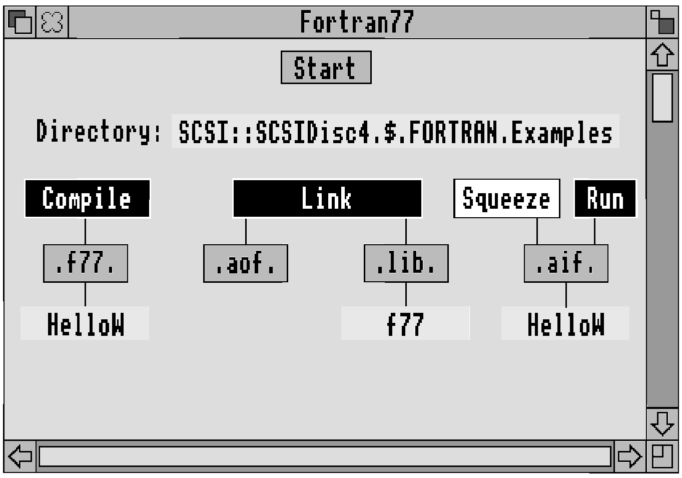
or

b) click the 'select' button over the !Fortran77 icon and drag the source text file(s) from an `f77` directory to the Fortran77 window.

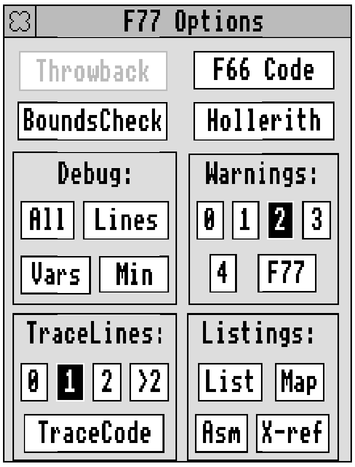


3 Drag any previously compiled object files to the Fortran77 window. The `f77` and `aof` directories must both be sub-directories of the same directory.

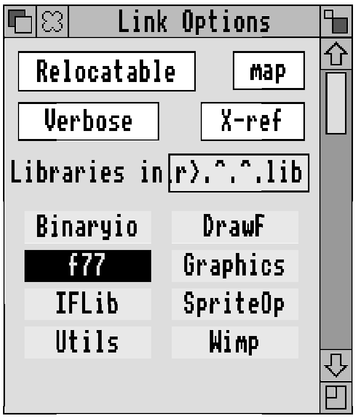
4 Click 'select' over the Compile, Link, Squeeze or Run icons to select the operations required.



5 Click 'menu' over the Compile icon to display the compiler options. Click 'select' to select an option. Click 'adjust' to de-select an option. Click 'select' over Throwback to select throwback of errors and provide on-line help. Do not select any of the debug options, see Appendix L.



6 Click 'menu' over the Link icon to display the linker options. Click 'select' to select an option. Click 'adjust' to de-select an option. Choose the libraries to be included by clicking 'select' over the appropriate library name.



7 Click 'menu' over the Run icon to enter command line arguments.

8 Click 'select' over the Start icon to carry out the selected operations.

9 Click 'select' over an item in a list of files to move it to the top (this is the way to change the scanning order of library files). Click 'adjust' over an item in a list of files to remove it.

If there are any compilation or link errors, they are written to the file `err`.*prog* (where *prog* is the name of the source text file). The details are displayed by !Edit or !SrcEdit (if RISC OS has 'seen' them); similarly, `asm`.*prog*, `lis`.*prog*, `map`.*prog* are created if the corresponding option has been selected.

Temporary files used during compiling and linking are stored on a RAM disc if possible, otherwise they are in a directory `$.tmp`. These are deleted when they are no longer required.

Extensions to the Standard

The FORTRAN 77 compiler offers several extensions to the standard. Further extensions concerning input/output are described in the next chapter.

Hexadecimal Constants

The compiler allows hexadecimal constants to be used and has the following form

```
?<type> <digits>
```

type is a letter, specifying the type of the constant. It must be one of I, R, D, C, L, H, or Q (for INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER and COMPLEX*16, 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 memory. With other constants, the most significant byte is given first. If the type of the constant is REAL, DOUBLE PRECISION, COMPLEX or COMPLEX*16, the number of bytes must match the size of the item in memory (4, 8 or 16); for INTEGER and LOGICAL constants, there may be fewer bytes.

Example

```
CHARACTER WINDOW * (*)
PARAMETER (WINDOW = ?H1CO5141EOC)
J = ?I1234
```

WINDOW consists of the bytes 1C 05 14 1E OC, and J is set to the decimal value 4660.

Naming

The compiler converts all lower case letters (apart from FORMATs and CHARACTER constants) to upper-case when it reads the source text, so all statements, identifiers, etc may be in lower case. Names may be up to 255 characters long. However, there is no limit on the length of CHARACTER values.

Loops

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

```
l1 IF (.NOT. logical expr)   GOTO l2
    ...
    ...
    GOTO l1
```

```
l2
```

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.

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:

```
  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.

Random Number Generators

The compiler has two routines for random number generation:

```
 REAL FUNCTION RNDO1 ()
```

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.

INCLUDE Statement

An `INCLUDE` statement allows a file containing source text to be read in by the compiler at the point where the `INCLUDE` statement occurs. The syntax of the statement is

```
  INCLUDE `filename'
```

Line numbers in the `INCLUDE` file are not recorded in the object file and will, therefore, not appear in a backtrace. The correct line numbers are shown in the program listing and in the error messages.

Type Names

`REAL*8` may be used as an alternative to `DOUBLE PRECISION`. The type names `LOGICAL*4, INTEGER*4, REAL*4` and `COMPLEX*8` are alternatives to `LOGICAL, INTEGER, REAL`, and `COMPLEX`, respectively.

COMPLEX*16

A `COMPLEX*16` value consists of a pair of `DOUBLE PRECISION` numbers, representing the real and imaginary parts of a complex number. The rules for the use of `COMPLEX*16` are the same for COMPLEX, with a few exceptions

Combining a COMPLEX*16 with a REAL or COMPLEX gives a COMPLEX*16 result.

Combining a COMPLEX with a DOUBLE PRECISION gives a COMPLEX*16 result.

A complex constant containing a double precision value is a COMPLEX*16.

The intrinsic function DIMAG is used to extract the imaginary part of a COMPLEX*16. DCMPLX is used to convert to COMPLEX*16; it may have one or two arguments.

The rules for memory layout and EQUIVALENCE of COMPLEX*16 are the same as for COMPLEX, except that the individual parts are DOUBLE PRECISION, rather than REAL.

There are new specific names for intrinsic functions with COMPLEX*16 arguments. These are

Generic	Specific
ABS	CDABS
CONJG	DCONJG
SQRT	CDSQRT
EXP	CDEXP
LOG	CDLOG
SIN	CDSIN
COS	CDCOS

Bit Manipulation Functions

There are eight intrinsic functions concerned with bit manipulation on INTEGER arguments. These are

IAND(I,J)	logical and of I and J.
IOR(I,J)	logical or of I and J.
IEOR(I,J)	logical exclusive or of I and J.
NOT(I)	logical complement of I.
ISHFT(I,J)	return I shifted left J places if J is positive or shifted right J places if J is negative. The result is undefined if J is not in the range - 32 to +32. Bits shifted out at the end are lost; zeros are introduced at the other end.
IBSET(I,J)	return I with bit J set to one. Bit zero is the least significant bit. The result is undefined if J is not in the range 0-31.
IBCLR(I,J)	return I with bit J set to zero.
BTEST(I,J)	test bit J of I and return a LOGICAL result - .TRUE. if the bit is set and .FALSE. if it is clear.

Note that BTEST returns a LOGICAL result whilst the other functions return an INTEGER result.

Example

```
IF (BTEST(IX, 0)) ...
```

tests to see if IX is odd.

```
I=IAND(I, ?IFF)
```

clears all but the least significant byte of I.

```
I=ISHFT(J,-24)
```

extracts the most significant byte of J.

Relaxed Rules for List-Directed Input

When reading a complex value using list-directed (free format) input, an integer or real constant can be given - the imaginary part of the value is set to zero.

When reading a character value, if the constant

does not start with a quote
is contained on a single record
does not contain an embedded space, comma or / character
does not start with digits followed by a *,

then the delimiting quotes may be omitted and embedded quotes are not doubled.

RISC OS Interface Routines

The small utility library IFExt, see Appendix I, contains alternative routines to those listed in this section.

The small utility library IFLib, see Appendix J, includes routines to return the addresses of variables, make SWI calls and read and write memory.

The FORTRAN run-time library contains the following routines to interface to the operating system. Examples illustrating the use of these routines are included in the `FORTRAN.Examples.General` directory.

OSBYTE

Purpose
To make OS_Byte calls which do not return any values.

Example
`CALL OSBYTE (IFUNC, IARG1, IARG2)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

OSBYTE1

Purpose
To make OS_Byte calls that return one value.

Example
`CALL OSBYTE1 (IFUNC, IARG1, IARG2, IRES1)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

Results
`IRES1` (integer) - `IRES1 = R1`

OSBYTE2

Purpose
To make OS_Byte calls that return two values.

Example
`CALL OSBYTE2 (IFUNC, IARG1, IARG2, IRES1, IRES2)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

Results
`IRES1` (integer) - `IRES1 = R1`
`IRES2` (integer) - `IRES2 = R2`

OSWORD

Purpose
To make OS_Word calls.

Example
```
CALL OSWORD (ICODE, IARRAY)
```

Parameters
`ICODE` (integer) - `R0 = ICODE`
`IARRAY` (integer array) - `R1` = pointer to `IARRAY` (one dimensional integer array - the OS_Word parameter block)

Results
Placed in `IARRAY`.

OSCLI

Purpose
To make OS_CLI calls.

Example
```
LOGICAL FUNCTION OSCLI (STRING)
LOGICAL STATUS
STATUS = OSCLI (STRING)
```

Parameters
`STRING` (character) - command line terminated by a carriage return.

Results
`STATUS` (logical) - if the command is executed without error `STATUS = .TRUE.` or if an error occurs `STATUS = .FALSE.`

OSGETERROR

Purpose
To return the error number and error message immediately after OSCLI has returned with `STATUS = .FALSE.`

Example
```
CALL OSGETERROR (IERRNO, ERRSTR)
```

Parameters
None

Results
`IERRNO` (integer) - the error number
`ERRSTR` (character) - the error message

Input/Output

Unit Numbers and Files

A FORTRAN unit number is used to refer 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. Zero is equivalent to the asterisk and may only be used in sequential `READ`s and `WRITE`s. Note that the filing system limits the number of files that can be open simultaneously.

A unit may be associated with 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 associated with the given filename. Otherwise, the command line arguments are scanned.

The format of the command line is

`command [`*`filename1 filename2 ...`*`][`*`unitno1`*`=`*`filename`*`1` *`unitno2`*`=`*`filename2`*`]`

An optional list of filenames is followed by an optional list of assignments of unit numbers to file names. The initial list of unassigned filenames are associated with units numbers 1, 2, 3, etc. Each assigned filename is associated with the given unit number. All unassigned filenames must precede any assigned filenames.

Example

```
PROG ABC DEF
```

This associates the file `ABC` with unit number 1 and `DEF` with unit number 2.

```
PROG 10=RESULTS
```

This associates the file `RESULTS` with unit number 10.

```
PROG RESULTS1 32=RESULTS2 3=X
```

This associates `RESULTS1` with unit number 1, `RESULTS 2` with unit number 32, and `X` with unit number 3.

The two `*` units always refer to the screen and keyboard. Any units which are not associated with a file in an `OPEN` statement or through command line arguments also refer to the screen and keyboard.

The output stream to the screen can be redirected to output to a file using the standard RISC OS syntax `{>`*`filename`*`}` and the input stream from the keyboard can be redirected to input from a file using the standard RISC OS syntax `{<`*`filename`*`}`.

Any `OPEN` files are closed 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. When writing to files which are to be printed, the first character of each record is a carriage control code and does not form part of the data in the record. All units in the range 50-60 assume printer output format by default. On other units, specifying `` FORM=`PRINTER' `` in the first `OPEN` statement for the unit causes printer output format to be assumed for that unit. This is an extension to the standard.

Note that the printer output format does not imply output to any physical printer.

The carriage control codes which are recognised and their representation in files are described in the section Formatted I/O.

Sequential Files

OPEN and CLOSE

The OPEN statement for a sequential file does not specify whether the file is to be read from or written to. Therefore, the operating system is called to open the file when the first READ or WRITE statement is executed. An OPEN statement which refers to a non-existent file will not fail. The error will occur when a READ or WRITE is attempted and can be trapped by using ERR= in the READ or WRITE statement.

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

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

A sequential file may be used without an explicit OPEN statement. The file is opened when the first READ or WRITE statement which refers to its associated unit number is executed.

Formatted I/O

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 a file or terminal. All input records are assumed to be extended indefinitely with spaces. Therefore, an input format may refer to more characters than are actually present in the record. Input from a terminal uses normal line editing conventions including cursor copying. <CTRL D> (04) is treated as the end of file code which may be trapped by specifying END= in the READ statement.

For file input, the carriage return (0D) or line feed (0A) codes are recognised as record terminators. Form feed (0C) codes are ignored. If the record contains more than 512 data characters then the rest are ignored. The combination carriage return-line feed or line feed-carriage return is treated as a single record terminator.

When writing a record to a file or terminal, the carriage control code or codes are output first, followed by the data in the record. Trailing spaces in a record are not output.

The following carriage control codes 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 or 0) or CR (1 or +) is not output before the first record in a file. When a file is closed, a line feed code is output if the final record contained any data characters. This is done for all OPEN files when a program terminates normally.

When writing to a non-printer unit, each record is terminated by a new line. If a prompt line is required, a $ (or \) character may be included in the format. This suppresses the final new line and trailing spaces are not removed from the final line output. This may be used to generate interactive prompts.

```
WRITE (6, `(A$)') `Type an integer: '
```

The $ (or \) acts as a normal item (like /) and can occur anywhere in the format (except after any unused editing codes, since these will be skipped).

The following program illustrates interaction with a terminal

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

The CHAR function may be used to construct bytes for output as VDU control codes. The following will switch the screen to mode 3.

```
       WRITE (*, 3) CHAR(22), CHAR(3)
 3     FORMAT ($,2A)
```

Note that the $ format descriptor has been used to suppress the final new line.

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 pre-assigned units (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).

Unformatted I/O

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 of, or fewer, variables as illustrated below

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

I is read as 1 and J is read as 6. The first record contains 5x4 = 20 bytes of data, and the second 3x4 = 12 bytes of data.

Records of the same length could be achieved by padding all unformatted records, 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 internal file format of the record is the characters UF, a four byte count giving the number of data bytes, followed by the data bytes. The UF characters are used as a check that the file contains valid unformatted records. The two records written in the example above would contain the following bytes:

```
55 46 14 00 00 00                  U F    no of data bytes = 20

01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00  data

55 46 0C 00 00 00                  U F    no of data bytes = 12
```

```
06 00 00 00 07 00 00 00 08 00 00 00                          data
```

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 specifying the record length `RECL=` must be used for a direct access file. The record length is measured in bytes, and formatted records are padded with spaces to this length.

The internal file format of a direct access file is the characters DA followed by a four byte count giving the record length. It is permissible to `OPEN` a direct access file specifying a smaller record length than was given when the file was created. The maximum permitted record length for a formatted direct access `OPEN` is 512 bytes; there is no limit for unformatted files. If the file has been `OPEN`ed 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.

As it is possible both to read from and write to a direct access file, the operating system is called to open the file when the `OPEN` statement is executed, rather than being delayed until the first `READ` or `WRITE`, which occurs with an `OPEN` statement for a sequential file. Therefore, any errors which occur may be trapped by specifying `ERR=` in the `OPEN` statement.

The following program uses direct access to write to and read from a file.

```
      OPEN (42, ACCESS='DIRECT', FILE='EGDATA', 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 `'EGDATA'` used in the above example need not exist, but if it does, it must be a valid direct access file with a record length greater than or equal to 16.

OPEN and CLOSE

The `OPEN` and `CLOSE` statements have been discussed above. Specifying `STATUS = NEW` or `STATUS = OLD` in the `OPEN` statement has no effect.

INQUIRE

INQUIRE by unit

`EXIST=` returns `.TRUE.` if the unit is in the valid range. It is not possible to return accurate responses for `SEQUENTIAL=, DIRECT=, FORMATTED=` and `UNFORMATTED=.` `` `YES' `` is returned if the unit is currently being used for the relevant access type, otherwise `` `UNKNOWN' `` is returned. Note that `NAMED=` can only be used if `FILE=` was specified in the `OPEN` statement for the unit. Command line file assignments are not available to `INQUIRE.`

INQUIRE by file

If FILE= was specified in an OPEN statement for a unit (and not CLOSEd), information deduced from that association is returned (for example, DIRECT= is returned as `YES' if the file is open for direct access), and the file is assumed to exist. Otherwise, if the file exists, EXIST= returns .TRUE. and SEQUENTIAL=, DIRECT=, FORMATTED= and UNFORMATTED= return `UNKNOWN'.

BACKSPACE

BACKSPACE is not implemented.

ENDFILE

ENDFILE sets the end of file status and prevents further file access.

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 operating system is called to open the file when the first READ or WRITE statement is executed.

Format Decoding

Format specifications are decoded in a more liberal manner than as defined by the FORTRAN 77 standard.

Lower case

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).

Extraneous repeat counts

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

Edit descriptor separators

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

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 same case in the output form. If no exponent field width is specified then, except for G edit descriptors, the initial character of the descriptor is used in the same case in the output form.

If an exponent field width is given as zero, a field width of 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. A field width of zero implies the correct width for the data type being transferred; Z by itself is an abbreviation for Z0.

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.

Abbreviations

```
symbol  abbreviation
OP        P
1X        X
T1        T
TL1       TL
TR1       TR
AO        A
```

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 variable must not have a fractional part or a negative exponent.

$ and \ descriptors

A $ or \ descriptor in a format specification suppresses the final newline when writing to a non-printer file.

Graphics

FORTRAN programs can write control codes to the RISC OS VDU drivers to produce graphics. The CHAR function is used to convert an integer code to a character for output.

The basic form of WRITE statement to generate graphics is:

```
      WRITE(*, `($,10A)') CHAR(code1), CHAR(code 2), ...
```

or

```
      PRINT `($,10A)', CHAR(code1), CHAR(code2), ...
```

The WRITE statement uses the standard asterisk output unit. Any non-printer unit (1- 49) could be used instead. The repeat count in the format specification (10 in these examples) must not be less than the number of VDU codes in the list. The $ format descriptor must be used to suppress the final newline.

The format can be given as a character constant, as in the examples above, or in a separate statement.

```
      PRINT 100, CHAR(code1), CHAR(code2), ...

 100  FORMAT($,10A)
```

For example, to change to mode 12

```
      PRINT `($,2A)', CHAR(22), CHAR(12)
```

or to change the palette so that colour 1 refers to colour 6

```
      PRINT `($,6A)', CHAR(19), CHAR(1), CHAR(6),
     +                CHAR(0), CHAR(0), CHAR(0)
```

Most move and draw operations require a pair of 16-bit coordinates. These should be output as a pair of bytes. For example, the following subroutine provides a general PLOT command (VDU code 25)

```
      SUBROUTINE PLOT(TYPE, X, Y)
      INTEGER TYPE, X, Y
      PRINT ` ($,6A)', CHAR(25), CHAR(TYPE),
     +            CHAR(IAND(X,255)), CHAR(ISHFT(X,-8)),
     +            CHAR(IAND(Y,255)), CHAR(ISHFT(Y,-8))
      END
```

Move is a TYPE=4 and draw is a TYPE=5 call to the subroutine PLOT.

Errors and Debugging

Errors can be detected both by the compiler and by the run-time library. In addition to generating error messages the compiler may also generate warning messages which indicate that the program may not behave as anticipated. An example of this is using a variable that has not been declared. An example of a fault which is not detected by the compiler, but by the run-time library, is attempting to divide by zero.

Front End Error Messages

Errors detected by the compiler front end are of a different type from those detected by the code generator. Front end error messages are short, obvious statements indicating that the compiler has detected unacceptable syntax. These messages are self-explanatory. There are two classes of error.

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 incorrect FORTRAN statement

```
100  ERRONEOUS
```

then the message produced would 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. For example, information about missing labels is given at the end of the program unit, rather than where the non-existent label was referenced.

The distinction between these two types of error message has been made in order to show that errors do not necessarily occur at the line where the message is given.

Warning Messages

The `W` compilation option enables the compiler to generate warnings. These warnings are graded in severity from 1 (the most serious) to 4, and are useful if the program behaves in an unexpected way.

Level 1 warns, for example, of statements that will not be executed because they follow a `GOTO` statement and are unlabelled.

Level 2 warns of the use of extensions to standard FORTRAN 77. These extensions may give problems if the program it be re-compiled on another make of computer (eg an IBM PC).

Levels 3 and 4 warn of source text that conforms to the standard syntax but is of unusual style and, therefore, could possibly be a mistake.

The strict FORTRAN 77 option 7 is used to control warnings about language extensions. If unset, warnings are not generated. Otherwise, messages are generated if the warning level (Wn) is 2 (the default) or greater. Option 7 is unset by default so that the extensions may be used without generating messages, whatever the warning level.

Code Generator Error Messages

These are not always as explicit as front end error messages and are listed in Appendix A with a brief explanation of the most likely cause. As was the case with front end error messages, errors do necessarily occur at the line where the message is given.

Code Generator Limits

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

code size	2 Mbytes
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. Normally the code generator will run out of memory before this happens.

Run-time Errors

A program may compile and link but when it is run error messages are generated. These error messages are generated by the run-time library and have 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 a re-tracing of the steps which the 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 address 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.

```
++++ ERROR 1025: LD input data not INTEGER

    Routine              data area    line

    F77_INIT             &000100D8
    F77_I067             &00010000
    ERR2  &0000FF04      16
    ERR1  &0000F9B4      10
    F77_MAIN             &0000F9B0     6
```

In this example, the main program (with default name `F77_MAIN`) has called `ERR1`, which in turn has called `ERR2`, which has attempted to read an integer using list-directed input (`F77_I067` and `F77_INIT` are internal routines in the run-time library).

The call to `ERR1` in the main program was on line 6, the call to `ERR2` in `ERR1` was on line 10, etc. The appearance of line numbers in the backtrace is controlled by the compiler `L` option (level 1 is the default).

If a hardware trap occurs in a program compiled with a line number option level 1, it may not be possible to determine the exact line number.

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

    Routine              data area    line

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

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

Code 1000 errors

There are a number of simple run-time errors producing error messages which all have the same error number of 1000. These are listed in Appendix B.

Array and Substring Errors

There are two errors which may be generated by a program unit which has been compiled with the bound checking option

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

An illegal array subscript has been used.

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

An illegal substring has been used.

Input/Output Errors

I/O errors are those which may be trapped by the use of `END=` and `ERR=` specifiers in FORTRAN 77 statements. If these specifiers are not used, an error message and code are generated as described below. Otherwise, execution continues, with the error code available through the 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 I/O operation being attempted (which may be `OPEN`, `CLOSE`, `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 generated, 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.

Tracing

To specify that calls to special trace routines are to be included in the code, select the `T` option when compiling. These routines will cause trace information to be output when

entering the program unit

leaving the program unit

a labelled statement is about to be executed

the `THEN` clause of an `IF...THEN` or `ELSEIF...THEN` construct is about to be executed

the `ELSE` clause of an `IF...THEN` or `ELSEIF...THEN` construct is about to be executed

a `DO` statement is about to be executed another subprogram unit is about to be executed.

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 available.

The first of these, `HISTOR` (short for History), 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 second routine provided is `BACKTR` (short for Backtrace) which outputs 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 called after the backtrace information has been generated. Under RISC OS, all tracing output is sent to the screen or may be sent to a file using the SPOOL command.

Appendix A

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 memory size.

`array <name> has invalid size`
The size of the given array is negative or exceeds memory size.

`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 memory size.

`concatenation too long`
The result of a CHARACTER concatenation may exceed memory size.

`conversion to integer failed`
A REAL or DOUBLE PRECISION value is too large for conversion to an @xr

`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, and 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 memory.

`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 the 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`

`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. This error message would occur with the following program:

```
      CALL ABC(1.0)
      CALL ABC(2)
      END
```

Appendix B

Run-time Error Messages

Code 1000 errors

`<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.

`<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.

`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 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.

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

`Unformatted input record too short`
Input record does not contain sufficient data.

`mismatched use of ACCESS, RECL in OPEN`
`ACCESS=`DIRECT'` has been quoted in an `OPEN` which does not contain a `RECL` specifier, or

vice versa.

Input/Output Errors

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

1002 `invalid attribute`
Invalid attribute used in OPEN statement

1003 `duplicate use of filename`
The same filename 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 I/O error has been detected previously on this unit, and trapped with ERR=.

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 access not open for input`
The direct access file could not be opened for input (for example, file is write only).

1010 `direct access not open for output`
The direct access file could not be opened for output (for example, file is read only).

1011 `end of file on output`
An attempt has been made to write beyond the end of a sequential file.
(In practice, this will only occur with internal files).

1020 `invalid logical in input`
Formatted input file contains bad logical value.

1021 `invalid number in input`
Bad number (range or syntax) in formatted I, D, E, F, or G input.

1022 `Bad complex data`
Bad COMPLEX constant in list directed input.

1023 `LD repeat not integer`
Repeat count in list directed input is not valid.

1024 `LD input data not REAL`
Syntax or range error in REAL list directed input value.

1025 `LD input data not INTEGER`
Syntax or range error in INTEGER list directed input value.

1026 `LD input data not DP`
Syntax or range error in DOUBLE PRECISION list directed input value.

1027 `LD input data not LOGICAL`

Syntax error in `LOGICAL` list directed input value.

1028 `LD input data not COMPLEX`

Syntax or range error in `COMPLEX` list directed input value.

1029 `LD input data not CHARACTER`

Syntax error in `CHARACTER` list directed input value.

1030 `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.

2000 `not available`

`BACKSPACE` operation is not available.

2001 `bad unformatted record (message)`

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

2002 `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.

2003 `sequential open failed (message)`

The actual reason for the failure (for example, Bad name) is given in the brackets.

2004 `direct access open failed (message)`

The actual reason for the failure (for example, Bad name) is given in the brackets.

2005 `direct access IO failed (message)`

For example, attempt to read beyond the end of the file.

2006 `record length too large`

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

2007 `bad direct access file (message)`

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

2009 `bad command line syntax`

2010 `sequential write failed (message)`

I/O error on sequential output (for example, cannot extend)

Appendix C

The Front End - The f77fe Command

This reads FORTRAN 77 source text and converts it to a special intermediate form known as fcode. The front end has the options: `X, W, T, 6` and `7`. The default settings are `X0W2-T67`.

The front end has the following command format

```
f77fe [-from] filename [-to filename] [-list filename] [-opt options]
[-ver filename]
```

`-from` *filename*

The `-from` keyword specifies the *filename* of the FORTRAN 77 source text input file.

`-to` *filename*

The `-to` keyword specifies the *filename* of the fcode format output file. If this keyword is not used then no output is produced.

`-list` *filename*

The `-list` keyword specifies the *filename* of the list output file for a line numbered listing of the source text together with any error messages generated. If this keyword is not used no listing is produced and error messages are output to the screen.

`-opt` *options*

The `-opt` keyword specifies the *options*. The options T, 6 and 7 are enabled or disabled by preceeding them with + or -. The options W and X must be followed by a number. The options have the following meanings:

6 This option allows FORTRAN 66 source text to be compiled. Constructs which have a different meaning in FORTRAN 77 are interpreted according to the FORTRAN 66 definition. In particular:

DO loops will always execute at least once.

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

Non-CHARACTER array names are allowed as format specifiers.

When the FORTRAN 66 option 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 option is used:

```
CALL jim('abcd')
CALL jim(4habcd)
```

If the FORTRAN 66 option is used, 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 option was introduced to allow FORTRAN 66 programs to be compiled. It isstrongly recommended that new programs conform to the FORTRAN 77 standard.

T This option causes special trace routines to be include in the code (See the chapter Errors and Debugging).

Wn This option specifies the warning message level. n=0 suppresses all warnings ton=4 print all warnings (See the chapter Errors and Debugging).

Xn This option specifies the cross-reference listing width (18 or more for legibility). n=0 suppresses cross-referencing. The maximum value of n depends on where the listing is being sent (for example, the 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 the line number of the statement is given, as well as the lines on which the label is referenced.

7 This option is used to control warnings about the use of FORTRAN 77 language extensions. If it is not enabled, warnings are not generated. If it is enabled warnings are generated when the warning level (`W`*n*) is 2 (the default) or greater. Warnings are not enabled by default so the extensions may be used without warnings being generated whatever the warning level.

`-ver` *`filename`*

The `-ver` keyword specifies the *`filename`* of the output file for compiler and error messages generated. If the keyword is not used the messages are output to the screen.

`-help`

The `-help` keyword gives a summary of the keywords and arguments available.

Examples

`f77fe f77.prog -to tmp.fcode`

Compiles the source text in the file `f77.prog` to fcode format in the file `tmp.fcode`.

`f77fe f77.prog -ver x`

Compiles the source text in the file `f77.prog`, producing no fcode output, but with messages output to the file `x`.

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

Compiles the source text in the file `f77.prog` to fcode format in the file `tmp.fcode` and also outputs a source listing to the file `list.prog`.

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

Compiles the source text in the file `f77.prog` to fcode format in the file `tmp.fcode` with tracing calls included.

Appendix D

The Code Generator - The f77cg Command

This reads fcode format and generates aof format and/or assembler source text format. The code generator has the options: `L`, `B` and `H`. The option `6` may be used instead of `H`. The default settings are `L1-BH`.

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.

The code generator has the following command format

`f77cg [-fcode]` *`filename`* `[-to` *`filename`*`] [-asm` *`filename`*`] [-ver` *`filename`*`] [-map` *`filename`*`] [-opt` *`options`*`] [-debug` *`level`*`] -source` *`name`*`]`

`-fcode` *`filename`*
The `-fcode` keyword specifies the *`filename`* of the fcode format input file.

`-to` *`filename`*
The `-to` keyword specifies the *`filename`* of the aof format output file generated. If the keyword is not used then no aof format output file is generated.

`-asm` *`filename`*
The `-asm` keyword specifies the *`filename`* of the assembler source text format output file equivalent to the aof format generated. If the keyword is not used then no assembler source text file is produced.

`-ver` *`filename`*
The `-ver` keyword specifies the *`filename`* of the output file for compiler and error messages generated. If the keyword is not used the messages are output to the screen.

`-opt` *`options`*
The `-opt` keyword specifies the *`options`*. The options `B` and `H` are enabled or disabled by preceeding them with + or -. The option `L` must be followed by a number. The options have the following meanings:

`B` When enabled, bound checking code is included. Array or substring subscripts out of range will cause run-time errors.

`H` When enabled, Hollerith constants can be used in `DATA` statements to initialise non-character variables (for example, `INTEGER`).

`L`*n* The number following this option indicates the level of line numbering included in the code for backtrace purposes (see the chapter Errors and Debugging). The levels available are:

`0` no line numbering

`1` numbers lines containing subprogram calls

`2` numbers 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 program unit compiled with level 1, the backtrace system will not be able to determine the exact line number. A range of numbers will be given (for example, 100/106) and the error will be between them.

`-map` *`filename`*

The `-map` keyword specifies the *`filename`* of map output file. 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.

`-debug` *`level`*

The `-debug` keyword specifies the *`level`* of symbolic debugging information to be included in the aof format output file. The *`leve`*l must be one of the following

`none`	No information. This is the default.
`min`	Subroutine and function names only.
`vars`	Subroutine and function names, and variable name information.
`lines`	Subroutine and function names, and line number information.
`all`	Subroutine, function, variable and line information. `max` is an alternative synonym for `all`.

Normally, `none` is used for a working program and `all` for programs under development. The use of `all` increases the size of the program considerably and so should be avoided when not debugging. The intermediate levels can be used to provide some debugging information without increasing the size of the program to the same extent.

`-source` *`filename`*

The `-source` keyword specifies the *`filename`* of the original FORTRAN 77 source text file for inclusion in debugging information when the `-debug` keyword is used with a debugging level other than the default of none.

`-help`

The `-help` keyword gives a summary of the keywords and arguments available.

Examples

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

Generates the aof format file `aof.prog` from the fcode format file `tmp.fcode`.

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

Generates assembler source text format and outputs it to the screen (`vdu:`) from the fcode format file `tmp.fcode`.

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

Generates the aof format file `aof.prog` from the fcode format file `tmp.fcode` and sends map output to `map.prog`.

```
f77cg tmp.fcode -to aof.prog -opt B
```

Generates the aof format file `aof.prog` from the fcode format file `tmp.fcode` with bound checking code included.

```
f77cg tmp.fcode -to aof.prog -debug all -source f77.prog
```

Generates the aof format file `aof.prog` from the fcode format file `tmp.fcode` with full debugging information and with the name of the FORTRAN source text file specified as `f77.prog`.

Appendix E

The Linkers

Two linkers are supplied: the older linker oldlink, see Appendix F, and the newer linker newlink, see Appendix G.

The linkers combine a number of object files with library files to produce a single executable program.

Each of the object files must be in Acorn Object Format (aof) or Acorn Library Format (alf). A file may contain references to external symbols (procedure and variable names) which the linker attempts to resolve by searching for definitions in the other files.

Usually, at least one library file will be specified. A library is a collection of Acorn Object Format files stored in a single Acorn Library Format file.

Libraries differ from object files in the way that the linker searches them. Object files are searched only once when the linker attempts to resolve external references. Libraries are searched as many times as necessary. If a required symbol is found in one of the component files of the library then the whole component file is incorporated in the output file.

Two common errors which occur during linking are caused by unresolved and multiple references.

In the first case, a symbol has been referenced in a file (whose name is given in the error), but there is no corresponding definition of the symbol. This is usually caused by the omission of a required object or library file, or the mis-spelling of a name in the original source program.

In the second case a clash of names occurs. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file. The version of the procedure used in any situation is the one local to the reference to it.

Wildcards can be used in the filenames. These will be expanded into the list of files matching the specification. For example, the name aof.bas* might be expanded into aof.basmain aof.basexpr and aof.bascmd.

Predefined Linker Symbols

There are several symbols which the linker knows about independently of any of its input files. These start with the string `Image$$` and, along with all other external names containing $$, are reserved by Acorn.

The symbols are:

`Image$$RO$$Base`	Address of the start of the read-only program area
`Image$$RO$$Limit`	Address of the byte beyond the end of program area
`Image$$ZI$$Base`	Address of the start of run-time zero-initialised area
`Image$$ZI$$Limit`	Address of the byte beyond the zero-initialised area
`Image$$RW$$Base`	Address of the start of the read/write (data) area
`Image$$RW$$Limit`	Address of the byte beyond the end of the data area

Although it will often be the case, it cannot be guaranteed that the end of the read- only area corresponds to the start of the read/write area.

These symbols can be imported as relocatable addresses by assembly language routines that might need them.

Note that programs can reside in read/write areas, as they sometimes contain local writable data (eg self modifying code), and it is possible to have read-only data (eg floating-point constants and string literals).

The linker joins all areas (from all input files) with the same name and attributes together to form a single area. It then creates the two symbols `name$$Base` and `name$$Limit` to mark the start and end of the area. It is an error for two areas to have the same name but different attributes.

Appendix F

The Older Linker - The oldlink Command

Acorn Object Format Linker ARM/Arthur(AIF) 595/M

To ensure that the older version of the linker, `oldlink`, is used the lines in the `FORTRAN.!Fortran77.!Run` obey file should read as follows:-

```
| Location of Command Line Commands
| --------------------------------
|
| Include the following line to ensure that the older version of the linker
| is used by the f77, f77lk, linkf77, d77, df77lk and dlinkf77 commands.

If "<F77$Running>" = "" Then Set Run$Path <Run$Path>,<Fortran77$Dir>.Execlib.OldLink.

| Include the following line to ensure that the newer version of the linker
| is used by the f77, f77lk, linkf77, d77, df77lk and dlinkf77 commands.

|If "<F77$Running>" = "" Then Set Run$Path <Run$Path>,<Fortran77$Dir>.Execlib.NewLink.
```

Command format:

```
oldlink -output filename [options] objectfile1, objectfile2 ...
oldlink -output filename [options] -via viafile
```

General options:
Capitals are used to denote the abbreviated form of the keyword.

-`Output` *`filename`*
Specifies the name of the output file as *`filename`*. The `f77link` command can be used to check for unresolved references in object files by specifying the file name as the device `null:`. The linked output will be discarded.

-`Dbug`
An output file is produced which can be used with the Acorn Symbolic Debugger `asd`.

-`Verbose`
Gives information as files are linked.

-`VIA` *`filename`*
The object and library files listed in the text file *`filename`* are linked. Note that this option cannot be used on computers fitted with a StrongARM processor.

Special options:

-`Case`	Ignore case when symbol matching
-`Base` *n*	Specify base of image (prefix 'n' with & for hex; postfix with k for $*2^{10}$, m for $*2^{20}$)
-`Relocatable`	Relocatable AIF

Note that -`Dbug` and -`Relocatable` are mutually exclusive options.

Appendix G

The Newer Linker link Command

ARM Linker Version 5.06 (Acorn Computers Ltd) [Jan 11 1995]

To ensure that the newer version of the linker, `newlink`, is used the lines in the `FORTRAN.!Fortran77.!Run` obey file should read as follows:-

```
| Location of Command Line Commands
| --------------------------------
|
| Include the following line to ensure that the older version of the linker
| is used by the f77, f77lk, linkf77, d77, df77lk and linkf77 commands.

|If "<F77$Running>" = "" Then Set Run$Path <Run$Path>,<Fortran77$Dir>.Execlib.OldLink.

| Include the following line to ensure that the newer version of the linker
| is used by the f77, f77lk, linkf77, d77, df77lk and linkf77 commands.

If "<F77$Running>" = "" Then Set Run$Path <Run$Path>,<Fortran77$Dir>.Execlib.NewLink.
```

Command format:
```
newlink -output filename [options] objectfile1, objectfile2 ...
newlink -output filename [options]  -via viafile
```

General options
Capitals are used to denote the abbreviated form of the keyword.

-`Output` *`filename`*
Specifies the name of the output file as *`filename.`* The `newlink` command can be used to check for unresolved references in object files by specifying the file name as the device `null`:. The linked output will be discarded.

-`Debug`
An output file is produced which can be used with the Desktop Debugger !DDT (see Appendix L).

-`ERRORS` *`filename`*
Diagnostic information is output to the file *`filename.`*

-`LIST` *`filename`*
Map and Xref information is output to the file *`filename`*, not `stdout`

-`MAP`
Prints an area map to the standard output

-`Symbols` *`filename`*
Symbol definitions are output to the file *`filename`*

-`Verbose`
Gives information as files are linked..

-VIA *`filename`*
The object and library files listed in the text file *`filename`* are linked.

-`Xref`
Prints an area cross-reference list to the standard output.

Output options

-`AIF`	Absolute AIF (the default)

-AIF - Relocatable	Relocatable AIF
-AIF - R -Workspace nnn	Self-moving AIF
-AOF	Partially linked AOF
-BIN	Plain binary
-BIN -AIF	Plain binary described by a prepended AIF header
-IHF	Intellec Hex Formay (readable text)
-SPLIT	Output RO and RW sections to separate files (-BIN, -IHF)
-SHL *filename*	Shared-library + stub, as described in *filename*
-SHL *filename* -REENTrant	Shared-library + reentrant stub
-RMF	RISC OS Module
-OVerlay *filename*	Overlaid image as described in *filename*

Special options

-R0-base n	
-Base n	Specify base of image
-RW-base n	
-DATA n	Specify separate base for image's data
-Entry n	Specify entry address
-Entry n+obj (area)	Specify entry as offset within area (prefix n with & or 0x for hex: postfix with K for $*2^{10}$, M for $*2^{20}$)
-Case	Ignore case when symbol matching
-MATCH n	Set last-gasp symbol matching option
-FIRST obj (area)	Place area from object obj first in the output image
-LAST obj (area)	PLace area from object obj last...
-NOUNUSEDareas	Do not eliminate AREAs unreachable from the AREA containing the entry point (AIF images only)
-Unresolved sym	Make all unresolved references refer to sym
-C++	Support C++ external naming conventions

Appendix H

The f77, f77lk and linkf77 Commands

In order ot use the f77, f77lk and linkf77 commands the currently selected directory must be set to the directory which contains the following directories

`f77` contains FORTRAN source text files
`aof` contains Acorn Object Format files for subsequent linking
`aif` contains executable Acorn Image Format files
o contains Acorn Object Format files for subsequent linking when a newer version of the linker is used (see Appendix G).

The version of the linker used, either the older (`pldlink`) or newer (`newlink`), can be selected by editing the `FORTRAN.!Fortran77.!Boot` obey file as shown in Appendices F and G.

The f77 Command
`f77` *`filename`*
The `f77` command combines the `f77fe` and `f77cg` commands to compile the file `f77.`*`filename`* using the default options.

The f77lk Command
`f77lk` *`filename`*
The `f77lk` command combines the `f77fe` and `f77cg` commands to compile the file `f77.`*`filename`* using the default options. It then links the resulting object file with the `IFExt`, `IFLib` and `f77` libraries to produce the program `aif.`*`filename`*.

The linkf77 Command
`linkf77` *`filename`*
The `linkf77` command links the object file *`filename`* with the `IFExt`, `IFLib` and `f77` libraries to produce the program `aif.`*`filename`*.

The df77, df77lk and dlinkf77 Commands

The df77, df77lk and dlinkf77 commands are similar to the f77, f77lk and linkf77 commands but include the -debug all rather than the default options.

Appendix I

The IFExt Utility Library

This is a small library which includes routines to make OS_Byte, OS_Word and OS_CLI calls. Examples illustrating the use of the routines are included in the `FORTRAN.Examples.IFExt` directory.

IFOSBYTE

Purpose
To make OS_Byte calls which do not return any values.

Example
`CALL IFOSBYTE(IFUNC, IARG1, IARG2)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

IFOSBYTE1

Purpose
To make OS_Byte calls that return one value.

Example
`CALL IFOSBYTE1(IFUNC, IARG1, IARG2, IRES1)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

Results
`IRES1` (integer) - `IRES1 = R1`

IFOSBYTE2

Purpose
To make OS_Byte calls that return two values.

Example
`CALL IFOSBYTE2(IFUNC, IARG1, IARG2, IRES1, IRES2)`

Parameters
`IFUNC` (integer) - `R0 = IFUNC`
`IARG1` (integer) - `R1 = IARG1`
`IARG2` (integer) - `R2 = IARG2`

Results
`IRES1` (integer) - `IRES1 = R1`
`IRES2` (integer) - `IRES2 = R2`

IFOSWORD

Purpose
To make OS_Word calls.

Example
`CALL IFOSWORD(ICODE, IARRAY)`

Parameters
ICODE (integer) - R0 = ICODE
IARRAY (integer array) - R1 = pointer to IARRAY (one dimensional integer array - the OS_Word parameter block)

Results
Placed in IARRAY.

IFOSCLI

Purpose
To make OS_CLI calls.

Example
```
LOGICAL FUNCTION IFOSCLI (STRING)
LOGICAL STATUS
STATUS = IFOSCLI(STRING)
```

Parameters
STRING (character) - command line terminated by a carriage return.

Results
STATUS (logical) - if the command is executed without error STATUS = .TRUE. or if an error occurs STATUS = .FALSE.

IFOSGETERROR

Purpose
To return the error number and error message immediately after OSCLI has returned with STATUS = .FALSE.

Example
```
CALL IFOSGETERROR(IERRNO, ERRSTR)
```

Parameters
None

Results
IERRNO (integer) - the error number
ERRSTR (character) - the error

IFFILEEXISTS

Purpose
To check that a file exists

Example
```
LOGICAL FUNCTION IFFILEEXISTS (STRING)
LOGICAL EXISTS
EXISTS=IFFILEEXISTS(FILENAME)
```

Parameters
FILENAME (character)

Results
EXISTS (logical) - if the file exists EXISTS = .TRUE.
or if the file does not exist EXISTS = .FALSE.

Appendix J

The IFLib Utility Library

This is a small library which includes routines to return the addresses of variables, make SWI calls and read and write memory. Examples illustrating the use of the routines are included in the `FORTRAN.Examples.IFLib` directory.

IFADR

Purpose
To return the address of an integer variable

Example
`iadr=IFADR(inum)`

Parameters
`inum` (integer)

Results
`iadr` (integer) - address of variable `inum`

IFADRF

Purpose
To return the address of a real (single precision floating point) variable

Example
```
REAL fnum
iadrf=IFADRF(fnum)
```

Parameters
`fnum` (real)

Results
`iadrf` (integer) - address of variable `fnum`

IFADRD

Purpose
To return the address of a real (double precision floating point) variable

Example
```
DOUBLE PRECISION dnum
iadrd=IFADRD(dnum)
```

Parameters
`dnum` (double)

Results
`iadrd` (integer) - address of variable `dnum`

IFADRC

Purpose
To return the address of the character descriptor which contains the address of the character string in the first word and its length in the second

Example
```
CHARACTER string*3
iadrc=IFADRC(string)
```

Parameters

`string` (character)

Results
`iadrc` (integer) - address of character descriptor

IFADRL

Purpose
To return the address of an logical variable

Example
```
LOGICAL status
iadr=IFADRL(status)
```

Parameters
`status` (logical)

Results
`iadr` (integer) - address of variable `status`

IFADRCMPLX

Purpose
To return the address of a complex variable

Example
```
COMPLEX voltage
iadr=IFADRCMPLX(voltage)
```

Parameters
`voltage` (complex)

Results
`iadr` (integer) - address of variable `voltage`

IFADRCMPLX16

Purpose
To return the address of a double precision complex variable

Example
```
COMPLEX*16 current
iadr=IFADRCMPLX16(current)
```

Parameters
`current` (double precision complex)

Results
`iadr` (integer) - address of variable `current`

IFADRS

Purpose
To return the address of the character string

Example
```
iadrs=IFADRS(string)
```

Parameters
`string` (character)

Results
`iadrs` (integer) - address of character `string`

IFQSWI

Purpose
To enable SWI's to be called from FORTRAN 77

Example
`ierror=IFQSWI(numswi,iregs)`

Parameters
`numswi` (integer) - SWI number.
Clear bit 17 (&00020000) to abort on error
Set bit 17 (&00020000) to return on error

`iregs` (integer array) - one dimensional array with subscripts in the range 0 to 9
`iregs(0)=R0`, `iregs(1)=R1`, etc

Results
`ierror` (integer) - if no error occurs ierror = 0
If bit 17 of `numswi` (&00020000) is clear and an error occurs the function does not return.
If bit 17 of `numswi` (&00020000) is set and an error occurs the function returns and `ierror` = address of a standard error block.

`iregs` (integer array) - one dimensional array with subscripts in the range 0 to 9. `iregs(0)=R0, iregs(1)=R1`, etc

IFQSWIX

Purpose
To enable SWI's to be called from FORTRAN 77

Example
`ierror=IFQSWIX(numswi,iregs)`

Parameters
`numswi` (integer) - SWI number (the function sets bit 17 (&00020000) to return on error)

`iregs` (integer array) - one dimensional array with subscripts in the range 0 to 9
`iregs(0)=R0`, `iregs(1)=R1`, etc

Results
`ierror` (integer) - if no error occurs ierror = 0
If an error occurs the function returns and `ierror` = address of a standard error block.

`iregs` (integer array) - one dimensional array with subscripts in the range 0 to 9.
`iregs(0)=R0`, `iregs(1)=R1`, etc

IFRDB

Purpose
To return the byte stored at the given address and return it as an integer

Example
`inum = IFRDB(iaddress)`

Parameters
`iaddress` (integer)

Results
`inum` (integer) - byte stored at `iaddress`

IFWRB

Purpose
To store a byte at the given address

Example
`CALL IFWRB(inum,iaddress)`

Parameters
`inum` (integer) - byte to be stored at `iaddress`
`iaddress` (integer)

IFRDI/IFRDW

Purpose
To return the integer/word stored at the given address. Note that the address must be on a word boundary and is in the valid address range.

Example
`inum = IFRDI(iaddress)`
or
`inum = IFRDW(iaddress)`

Parameters
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

Results
`inum` (integer) - integer/word stored at `iaddress`

IFWRI/IFWRW

Purpose
To store an integer/word at the given address. Note that the address must be on a word boundary and is in the valid address range.

Example
`CALL IFWRI(inum,iaddress)`
or
`CALL IFWRW(inum,iaddress)`

Parameters
`inum` (integer) - integer/word to be stored at `iaddress`
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

IFHWRDB

Purpose
To return the byte stored at the given hardware address and return it as an integer. Note that the function does not check that the address is in the valid hardware address range.

Example
`inum = IFHWRDB(iaddress)`

Parameters
`iaddress` (integer)

Results
`inum` (integer) - byte stored at `iaddress`

IFHWWRB

Purpose
To store a byte at the given address. Note that the function does not check that the address is in the valid hardware address range.

Example
`CALL IFHWWRB(inum,iaddress)`

Parameters
`inum` (integer) - byte to be stored at `iaddress`
`iaddress` (integer)

IFHWRD16

Purpose
To return the 16 bits stored at the given hardware address and return them as an integer. Note that the address must be on a word boundary and be in the valid address range.

Example
`inum = IFHWRD16(iaddress)`

Parameters
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

Results
`inum` (integer) - 16 bits stored at `iaddress`

IFHWWR16

Purpose
To store 16 bits at the given address. Note that the address must be on a word boundary and be in the valid address range.

Example
`CALL IFHWWR16(inum,iaddress)`

Parameters
`inum` (integer) - 16 bits to be stored at `iaddress`
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

IFHWRDW

Purpose
To return the word (32 bits) stored at the given hardware address and return it as an integer. Note that the address must be on a word boundary and be in the valid address range.

Example
`inum = IFHWRDW(iaddress)`

Parameters
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

Results
`inum` (integer) - word stored at `iaddress`

IFHWWRW

Purpose
To store a word (32 bits) at the given address. Note that the address must be on a word boundary and be in the valid address range.

Example
`CALL IFHWWRW(inum,iaddress)`

Parameters
`inum` (integer) - word to be stored at `iaddress`
`iaddress` (integer) - must be on a word boundary ie be a multiple of 4

IFSWI

Purpose

To enable SWI's to be called from FORTRAN 77

Example

`ierror=IFSWI(numswi,iregsin,iregsout,iflags)`

Parameters

`numswi` (integer) - SWI number.
  Clear bit 17 (&00020000) to abort on error
  Set bit 17 (&00020000) to return on error

`iregsin` (integer array) - one dimensional array with subscripts in the range 0 to 9
  `iregsin(0)=R0`, `iregsin(1)=R1`, etc

Results

`ierror` (integer) - if no error occurs ierror = 0
  If bit 17 of `numswi` (&00020000) is clear and an error occurs the function does not return.
  If bit 17 of `numswi` (&00020000) is set and an error occurs the function returns and ierror = address of a standard error block.

`iregsout` (integer array) - one dimensional array with subscripts in the range 0 to 9.
  `iregsout(0)=R0`, `iregsout(1)=R1`, etc

`iflags` (integer) - processor flag bits
  bit0 = V flag bit1 = C flag bit2 = Z flag bit3 = N flag

IFSWIX

Purpose

To enable SWI's to be called from FORTRAN 77

Example

`ierror=IFSWIX(numswi,iregsin,iregsout,iflags)`

Parameters

`numswi` (integer) - SWI number (the function sets bit 17 (&00020000) to return on error)

`iregsin` (integer array) - one dimensional array with subscripts in the range 0 to 9
  `iregsin(0)=R0`, `iregsin(1)=R1`, etc

Results

`ierror` (integer) - if no error occurs `ierror` = 0
  If an error occurs the function returns and `ierror =` address of a standard error block.

`iregsout` (integer array) - one dimensional array with subscripts in the range 0 to 9.
  `iregsout(0)=R0`, `iregsout(1)=R1`, etc

`iflags` (integer) - processor flag bits
  bit0 = V flag bit1 = C flag bit2 = Z flag bit3 = N flag

Appendix K

Calling Functions and Subroutines Written in Assembler from FORTRAN

The FORTRAN compiler does not conform to the ARM Procedure Call Standard (APCS-R). Only one argument is passed on calling, register variables `v1-v6` and `f4-f7` are not preserved and the register binding of the APCS-R is not used.

Register Conventions

The register binding used is

`r0`	Pointer to a list of the addresses of the arguments given on calling.
`r1-r9`	Scratch registers.
`fp (r10)`	Frame pointer (used as a pointer to a list of the addresses of the arguments given on calling within the assembler routine).
`sp (r12)`	Stack pointer.
`sb (r13)`	Static base (used to refer to local data within the assembler routine).

Argument Lists

Every call in FORTRAN passes one argument in `r0`. This is a pointer to a list of the addresses of the arguments given in the call (every argument in FORTRAN is passed by reference). Thus the address of the first argument is at `[r0,#0]`, the second at `[r0,#4]`, etc. Normally, the address of the list is copied to `fp (r10)`, which is preserved by calls. If an assembler routine does not call any other routines, the address of the list can be left in `r0`.

For a `CHARACTER` argument, the address in the argument list does not point directly to the data. It points at a character descriptor, which is a two-word block containing the address of the character string in its first word and its length in the second. For example, if the third argument in a call is a `CHARACTER` value, the following loads its address into `r1` and its length into `r2`

```
LDR      r1,[r0,#8]          ; Descriptor address
LDMIA    r1,{r1,r2}          ; Address and length
```

Function Results

For non-`CHARACTER` functions, the address of the result is returned in `r0`. A `CHARACTER` function is implemented as a subroutine with the address of the result (a character descriptor) passed as an additional argument inserted before the other arguments (thus the first argument in the call appears as the second argument, etc).

A subroutine with alternate returns (`*'`s in the argument list) is implemented as an `INTEGER` function. The result should be zero for the main return, one for the first alternate return, two for the second, etc. The alternate return specifiers do not appear in the argument list.

Static Data

Static data for an assembler routine should be in a writable area and addressed using `sb (r13)` as this is preserved by calls.

Section Format

The code area of a FORTRAN-callable assembler routine should start with the routine name as a twelve-character string, padded with spaces. The address of the first byte following this name must be pushed on to the stack at the beginning of the routine. The code area should be named `F77$$Code` and have the attributes `CODE` and `READONLY`. The data area (if any) should be named `F77$$Data` and have the `DATA` attribute.

FORTRAN COMMON blocks should be defined as named AREAs with the COMMON and NOINIT attributes. An initialised COMMON block (equivalent to a BLOCK DATA subprogram) should be defined with the COMDEF (common definition) attribute. FORTRAN blank COMMON is given the name F77_BLANK.

The basic layout of a FORTRAN-callable assembler routine is

```
        TTL             "name"
;Registers
r0      RN              0
r1      RN              1
        ...
        ...
r9      RN              9
fp      RN              10
sp      RN              12
sb      RN              13
lr      RN              14
pc      RN              15

f0      FN              0
        ...
        ...
77      FN              7

;Data
        AREA            |F77$$Data|,DATA
        ... data declarations ...
;Code
        AREA            |F77$$Code|,CODE,READONLY

NAME    DCB             "ASMROUTINE  "  ; Name padded with spaces
                                        ; to 12 characters

DATAPTR DCD             |F77$$Data|     ; Address of data

        EXPORT ASMROUTINE

ASMROUTINE

        ADR              r1,NAME+12     ; Standard entry sequence
        STMFD sp!,{r1,fp,sb,lr}
        LDR              sb,DATAPTR     ; Address of data area
        MOV              fp,r0          ; Copy address of argument
list

        ... code ...

        LDMFD           sp!,{r1,fp,sb,pc} ; Standard exit sequence

        END
```

An Example FORTRAN-Callable Assembler Routine

```
;   s.IFADR
;   contains IFADR
;   07 June 1993 Version 0.00

;   Purpose
```

```
;   To return the address of an integer variable

;   Example
;   iadr=IFADR(inum)

;   Parameters

;   inum (integer)

;   Results

;   iadr (integer) - address of variable inum

; REGISTERS

; Use the RN directive to define ARM register names

r0      RN      0
r1      RN      1
r2      RN      2
r3      RN      3
r4      RN      4
r5      RN      5
r6      RN      6
r7      RN      7
r8      RN      8
r9      RN      9
r10     RN      10
fp      RN      10
r11     RN      11
r12     RN      12
sp      RN      12
r13     RN      13
sb      RN      13
r14     RN      14
lr      RN      r14
r15     RN      15
pc      RN      r15

; Use the FN directive to define floating point register names

f0      FN      0
f1      FN      1
f2      FN      2
f3      FN      3
f4      FN      4
f5      FN      5
f6      FN      6
f7      FN      7

; DATA
        AREA    |F77$$Data|,DATA

RESULT  %       4                  ; result location

; CODE
        AREA    |F77$$Code|,CODE,READONLY
NAME    DCB     "IFADR       "
```

```
DATAPTR DCD     |F77$$Data|

        EXPORT  IFADR

IFADR
        ADR     r1,NAME+12        ; standard entry sequence
        STMFD   sp!,{r1,fp,sb,lr}
        LDR     sb,DATAPTR
        MOV     fp,r0
                                  ; fp points to param block
        LDR     r1,[fp]           ; load address of integer
                                  ; sb points to data area
        STR     r1,[sb]           ; store address of integer

        LDMFD   sp!,{r1,fp,sb,pc} ; standard exit sequence

        END
```

An Example FORTRAN Program Which Calls the Assembler Routine

```
      PROGRAM Example
C     Demonstrates the use of IFADR

      int1 = 1

      iadrint1 = IFADR(int1)

      PRINT *,'Integer 1 is ',int1,' and its address is ',iadrint1

      STOP
      END
```

Assuming that the source text of the assembler routine is in the file `s.IFADR` and the source text of the FORTRAN program is in the file `f77.Example`, they should.be compiled and linked as follows

```
*f77 Example
*objasm s.IFADR o.IFADR -APCS NONE
*link -output aif.Example o.Example o.IFADR <F77$Lib>f77
```

Run the program by typing

```
*aif.Example
```

The following should be displayed

```
Integer 1 is 1 and its address is 106204

STOP
```

Appendix L

Notes on Using a Debugger

The version of the linker used by the df77, df77lk and dlinkf77 commands can be selected by editing the FORTRAN.!Fortran77.!Run obey file as shown in Appendices F and G.

The Acorn Symbolic Debugger

Run from the command line by typing

```
*asd aif.filename
```

This can only be used with programs linked by the older linker (oldlink). Edit the FORTRAN.!Fortran77.!Run obey file as shown in Appendix F and then use the df77, df77lk and dlinkf77 commands. Before using the Acorn Symbolic Debugger on a computer fitted with a StrongARM processor it is essential to turn the cache off. From the command line type

```
*cache off
```

To turn the cache on from the command line type

```
*cache on
```

The Desktop Debugger !DDT

Run from the command line by typing

```
*debugaif aif.filename
```

This can only be used with programs linked by the newer linker (newlink). Edit the FORTRAN.!Fortran77.!Run obey file as shown in Appendix G and then use the df77, df77lk and dlinkf77 commands.