

# ***Fortran 77***

# Legal matters

## *Copyright*

Copyright Fortran Friends

Neither the whole nor any part of the information contained herein, nor the products described in this manual may be adapted or reproduced in any material form without the prior written approval of Fortran Friends.

The products described in this manual and the products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the products and their use (including the information and particulars in this manual) are given in good faith. However no liability can be accepted for any loss or damage arising from the use of any information or particulars in this manual.

Please send any comments you have on this manual or the software it describes using the form on the [last page](#).

## *Trademarks*

ACORN, is the trademark of Acorn Computers Limited, now dissolved

PostScript is a trademark of Adobe Systems Inc.

## *Acknowledgements*

This manual was typeset on various RISC OS (Acorn) computers using TechWriter from Icon Technology Limited and Martin Würthner.

Interactions with the developers of the Fortran compiler (Codemist and [ROOL](#)) has been invaluable in producing this application.

# Contents

**Contents**

|  |                               |
|--|-------------------------------|
| <b>Introduction</b>                      | <a href="#"><u>7</u></a>      |
| Product software and documentation       | <a href="#"><u>8</u></a>      |
| Differences from Acorn Fortran release 2 |                               |
| Hardware requirements                    | <a href="#"><u>9</u></a>      |
| Typeface conventions                     |                               |
| <br><b>Installation</b>                  | <br><a href="#"><u>11</u></a> |
| Directory and file structure             | <a href="#"><u>12</u></a>     |
| !SrcEdit help files                      | <a href="#"><u>14</u></a>     |
| Testing your system                      | <a href="#"><u>15</u></a>     |
| Error messages                           | <a href="#"><u>15</u></a>     |
| <br><b>Getting Started</b>               | <br><a href="#"><u>17</u></a> |
| Tutorial                                 |                               |
| Writing a program                        |                               |
| Fortran Desktop Environment (FDE)        | <a href="#"><u>18</u></a>     |
| Compiler Options                         | <a href="#"><u>19</u></a>     |
| Pragmas                                  | <a href="#"><u>21</u></a>     |
| Compiling                                | <a href="#"><u>21</u></a>     |
| Errors in compilation                    |                               |
| Throwback                                |                               |
| On-line Fortran Help                     |                               |
| Linking                                  | <a href="#"><u>22</u></a>     |
| Compiling + linking + run                | <a href="#"><u>23</u></a>     |
| Squeeze                                  |                               |
| Libraries                                |                               |
| Link Errors                              |                               |
| Accessing applications libraries         | <a href="#"><u>23</u></a>     |
| Removing Libraries                       |                               |
| Library test programs                    |                               |
| Run options                              | <a href="#"><u>25</u></a>     |
| Working directory structure              |                               |
| Dragging files to the FDE                |                               |
| 'Check Dates'                            | <a href="#"><u>26</u></a>     |
| <br><b>Fortran System Library</b>        | <br><a href="#"><u>27</u></a> |
| Introduction and documentation           |                               |
| <br><b>Application Libraries</b>         | <br><a href="#"><u>31</u></a> |
| Introduction and documentation           |                               |
| Draw library                             | <a href="#"><u>33</u></a>     |
| Font library                             | <a href="#"><u>37</u></a>     |
| Graphics library                         | <a href="#"><u>39</u></a>     |

## **Contents**

## **Contents**

|  |                                |
|--|--------------------------------|
| Sprite Operations library                  | <a href="#"><u>43</u></a>      |
| Utilities library                          | <a href="#"><u>47</u></a>      |
| Wimp library                               | <a href="#"><u>49</u></a>      |
| tutorial on writing a Wimp program         | <a href="#"><u>55</u></a>      |
| <br><b>Reference Section</b>               | <br><a href="#"><u>63</u></a>  |
| Fortran Development Environment (FDE)      |                                |
| calling the compiler from the command line | <a href="#"><u>68</u></a>      |
| compiler options                           |                                |
| calling the linker from the command line   | <a href="#"><u>72</u></a>      |
| link options summary                       |                                |
| running jobs from the command line         | <a href="#"><u>72</u></a>      |
| extensions to the Fortran 77 standard      | <a href="#"><u>73</u></a>      |
| input/output extensions                    | <a href="#"><u>79</u></a>      |
| External File Formats                      | <a href="#"><u>82</u></a>      |
| inter-language calling                     | <a href="#"><u>84</u></a>      |
| ARM Procedure Calling Standard             | <a href="#"><u>88</u></a>      |
| <br><b>Debugger tutorial</b>               | <br><a href="#"><u>94</u></a>  |
| debugging with PRINT                       |                                |
| using !DDT                                 |                                |
| <br><b>Appendices</b>                      | <br><a href="#"><u>96</u></a>  |
| compiler error messages                    | <a href="#"><u>97</u></a>      |
| run-time error messages                    | <a href="#"><u>100</u></a>     |
| Input/Output Errors                        | <a href="#"><u>101</u></a>     |
| glossary of acronyms                       | <a href="#"><u>105</u></a>     |
| comment form                               | <a href="#"><u>105</u></a>     |
| <br><b>Index</b>                           | <br><a href="#"><u>106</u></a> |



# Introduction

### ***Introduction***

Over the last 50 years, Fortran has been the most popular computer language used by professional scientists and engineers. This manual describes a method of using the latest release of the Fortran 77 programming language for 32-bit RISC OS microcomputers.

This Fortran 77 compiler conforms to the American National Standard Programming Language X3.9-1978 (ANSI Fortran 77 standard), available from the British Standards Institute.

### ***Product Software***

The software in this product is in two parts:

- Fortran 77 specific new release should contain:
  - compiler (f77)
  - Fortran in the Desktop Environment (FDE) tool
  - Fortran system library (Fortlib)
  - application libraries
- Desktop tools, previously released with 'Desktop C' and 'Desktop Assembler':
  - !SrcEdit text editor for program source code (if you know !Edit this will appear very familiar)
  - !DDT (the debugging tool)
  - link
  - squeeze

The compiler, !SrcEdit, !DDT, link and squeeze may be obtained from RISC OS Open Limited ([ROOL](#)). FDE, the libraries and their sources are in the Fortran Friends distribution zipped directory, downloadable from their [website](#).

### ***Differences from release 2***

This release improves on previous versions in several important ways:

- it can be used from the desktop, using the Fortran Development Environment (FDE).
- it runs on 32 and 26-bit architecture computers.
- it conforms to the ARM Procedure Calling Standard ([APCS](#)) which means that routines written in other languages (e.g. 'C') can be called from Fortran and vice-versa.
- it has an optimising compiler which makes shorter and faster code.
- it allows recursive subroutine calls
- the FDE includes an extensive suite of application libraries.

### ***Product Documentation***

In addition to this manual on Fortran specific information and the Fortran desktop Development Environment, full details of the libraries are given in this free release in ASCII files which can be inspected or printed. These are stored as 'Help' libraries in !SrcEdit for interactive use (see the Help section of the Installation chapter for how to set this up). The source of a comprehensive set of examples are included to show how to use the application libraries.



### **System Hardware requirements**

Two megabytes of memory allows compilation of most reasonably sized programs. It is possible to develop small Fortran programs with 1Mbyte of memory and a single 800Kbyte floppy drive but more disc space is desirable, preferably on a hard drive, otherwise frequent disc swapping will be needed.

### **Assumed knowledge**

Some familiarity with the RISC OS desktop environment is assumed; you should know how to use the mouse, !Edit, the icon bar and file handling using directory windows. Instructions for using the desktop are in your 'RISC OS User Guide'.

**This manual is not a tutorial in Fortran.** A working knowledge of Fortran 77 is desirable, but you can learn some of the essentials from the tutorial and examples. If you need to know more, there are many suitable introductory texts e.g.

"Programming in Standard FORTRAN 77" by Balfour and Marwick, published by Heinemann Educational Books.

### **About this Manual**

This manual describes how to install and use Fortran Release 5 for 32-bit (or 26-bit) RISC OS microcomputers. It contains chapters on

- [Installation](#)
- [tutorial](#) to help you get started using the FDE
- a [Reference Section](#) containing:
  - [Compiler options](#)
  - [Link Options](#)
  - description of the [desktop](#) interface
  - [Fortran 77 Extensions](#) to the standard
  - External [File Formats](#)
  - inter-language calling and [APCS](#)
- libraries including:
  - [Fortran System Library](#) with extensions
  - [Application Libraries](#)
- Fortran tutorial using the debugger
- [Appendices](#)
  - [Compiler Error Messages](#)
  - [Run time errors](#)
  - [Glossary of Acronyms](#)
- [Index](#)

**Conventions used in this manual**

Various typefaces are used in this document:

- Descriptive text looks like this.
- Text you enter looks like this - Typewriter script . If it is the name of a generic object for which you have to enter the real name it is shown in italics, for example:  
`f77 -s filename.`  
If it is an optional parameter it is shown in square brackets,
- Text the computer prints is in italic :  
*Enter file name or <Return> to terminate.*
- Text in angled brackets, e.g. <Shift>, <Ctrl>, <Tab>, <Space> etc. means the key of that name.

The word 'type' means enter some text followed by <Return>, whereas 'press' means just push the key with no <Return>.

The mouse buttons are named from left to right, <Select>, <Menu> and <Adjust>.

The following shorthand is used for various mouse operations:

- 'Click' means press the mouse <Select> button once.
- 'Double click' means press it twice quickly.
- 'Drag' means hold <select> down over an icon until it shows a flashing outline, move the pointer, 'dragging' the outline, to another window and then release <select> thus 'dropping' the icon onto another window. This is frequently used for copying files; (hold down <Shift> to move a file rather than copying it.

# **Installation Guide**

## Introduction

This chapter describes how to set up a suitable file and directory structure for using Fortran on your RISC OS computer. There are two sources needed to build the structure:

- The files and directories within the free download from Fortran Friends which contains this manual (referred to as 'FDE' in these installation instructions);
- The DDE package from RISC OS Open Limited ([ROOL](#)) which you will have to buy. In this chapter it will be referred to as the 'DDE'.

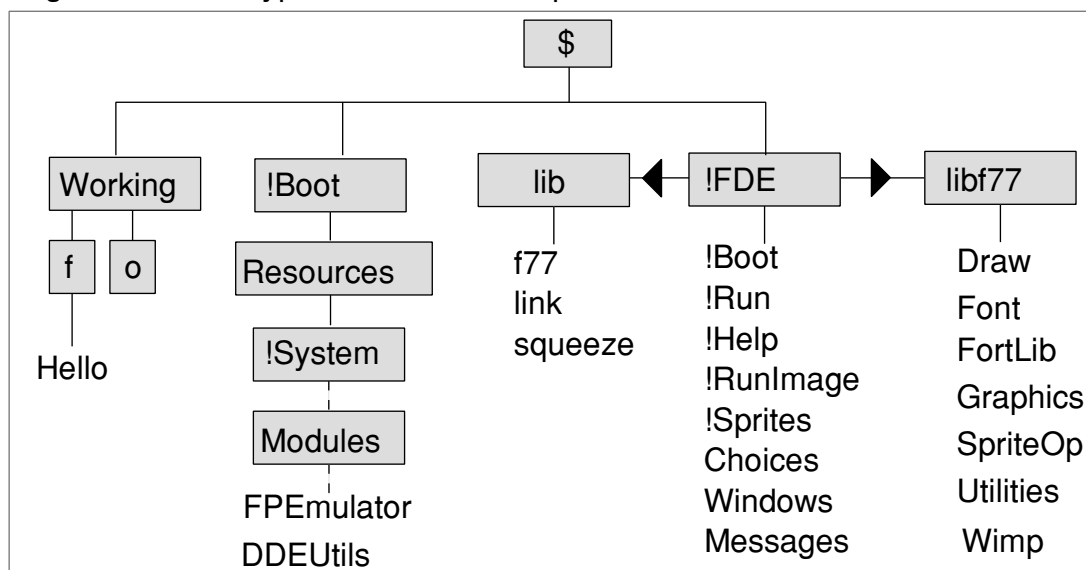
Within this DDE package you will need to extract parts from the directories:

- AcornC/C++ for the compiler, linker, squeeze and relocatable modules
- Apps\_DDE for !DDT and !SrcEdit  
*and*
- You should read Licence.Licence/pdf
- Other directories contain lots of stuff which are not actually needed here but which you might find useful elsewhere.

Using these sources you should be able to build up your Fortran environment as described in the next section.

## Directory and file structure

This diagram shows a typical hard disc setup:



The directories you will need are:

- ❑ !FDE - the Fortran front-end (from FDE) which must also include:
  - The usual application files + Choices, Messages and Windows,
  - A 'lib' sub-directory for the compiler, linker and squeeze. These are in the DDE (AcornC/C++.!SetPaths.Lib32.) but see the section on [editing !Run](#).
  - A 'libf77' sub-directory with the relocatable object libraries which *must include* the 'Fortlib' from FDE. The libraries are described in separate chapters of this manual and are all in the FDE.

The location of these sub-directories is defined in the !Run file of !FDE and could be stored anywhere but will then need changes to the [!Run file](#)

- ❑ Working directories for your programs containing:
  - f a sub-directory for your Fortran source files
  - o an optional sub-directory for the compiled object files
  - executable image files
- ❑ Also you will need to load the following applications from the DDE:
  - !SrcEdit the editor for Fortran source text files including separate help files for each library; you can find it in the DDE in the directory 'Apps.DDE'; it would be best to copy this to a private area because you will need to increment the help files.
  - !DDT the Desktop Debugging Tool which is also in 'Apps.DDE'; it is only needed for a debug session.
- ❑ Updates to your !System application containing the relocatable system modules needed by the applications above. You can update the !System in the Boot structure of RISC OS from the DDE in 'AcornC/C++.Developer.!System' by clicking on your !Boot and then System and following the instructions. The following modules should then be available:
  - DDEUtils a module used for 'throwback' of compiler error messages to !SrcEdit; this is probably already in your modules in !Boot.
  - DDT Desktop Debugging Tool module
  - FPEmulator The floating point emulator module

!DDT and the DDT module are only needed if you want to use the symbolic debugger.

### ***Editing !Run***

You may want to edit the !Run file of the !FDE application so the

'Set F77lib\$Dir ..' command points to the directory of Fortran libraries and the 'Set F77cl\$Dir ..' points to the directory containing the compiler, linker, etc. E.g. you can make it point directly to AcornC/C++.!SetPaths.Lib32 if you have loaded this to your hard disc by replacing this line with a line like:

```
'Set F77cl$Dir Filingsystem::Disc.$.AcornC/C++.!SetPaths.Lib32'
```

You will then not need the !FDE.lib directory

You will almost certainly need to alter the line pointing to your installation of !SrcEdit.

### Installing 'help' files

There are on-line help files in the directory: '!SrcEdit.help' in the FDE which are used to display helpful information when you select a keyword and press <F1>. Copy the !SrcEdit from the FDE on top of the !SrcEdit you extracted from the DDE.

!SrcEdit has pointers in '!SrcEdit.choices.languages' which contains three-line entries for each help library. For Example:

*Fortran*

*none*

*<SrcEdit\$dir>.help.HelpF77*

which is the entry for the standard Fortran keywords stored in the file

'!SrcEdit.help.HelpF77'. You may remove the help files to save space but should probably also remove the corresponding three-line entries in the

'!SrcEdit.choices.languages' file. Be careful not to lose any of the entries already in the downloaded '!SrcEdit.choices.languages' or any of the files already in '!SrcEdit.help'.

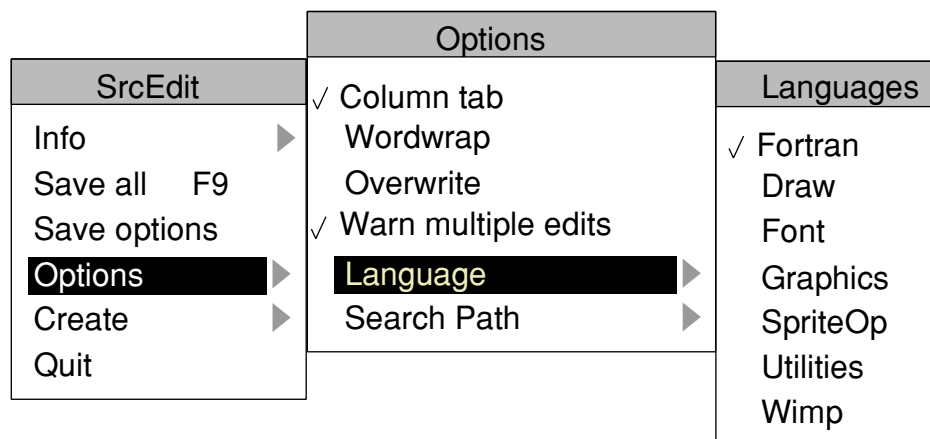
### Concatenating the help files

The help information on each library is contained in a separate file. It is possible to join these files of help information together and hence have less entries in the 'Languages' column in the above menu. The disadvantage in this is that !SrcEdit will take longer to find the information.

At the same time as joining help files together you would then have to edit the file '!SrcEdit.choices.languages' to correspond to your new collection of help files.

### On-line help from !SrcEdit

After installation, !SrcEdit is set up to give you on-line help on the entries in the libraries. To get help on a library, select it from the !SrcEdit menu: click <Menu> over the !SrcEdit icon on the icon bar and move the pointer over the arrow by the 'Options' entry and then over the arrow by the 'Language' entry to give you a menu tree like the following:



Click over the the library name in the last column for which you want help, initially 'Fortran', then make this your default language by clicking 'Save options' in the SrcEdit menu above.

### Testing your system

Try out your installation by following the '[Getting Started](#)' section in this manual.

#### Errors while installing and testing

| Error Message  | Problem defined   | Solution  |
|--|---|---|
| <i>Bad file name<br/>'squeeze'</i>                                   | The F77cl\$Dir has not been set in the !Run file  | Insert into the !Run file of the !FDE application, the line:<br>Set F77cl\$Dir xxxx<br>(where 'xxxx' is the full path name of the directory where the compiler (f77) linker and squeeze are kept. |
| <i>Directory [dirname] can not be found</i>                          | Either the directory containing the Fortran libraries, (F77Lib\$Dir), or that containing the compiler, (F77cl\$Dir) is missing. | Correct the variables pointing to these in the !Run file in the !FDE application  |
| <i>File '&lt;FDE\$Dir&gt;.Windows' not found</i>                     | there is not a file called 'Windows' in the !FDE application  | copy a new version of the file from the !FDE on the web   |
| <i>Filing system [xxxx] must be given a file name</i>                | The F77lib\$Dir has not been set in the !Run file   | Insert into the !Run file of the !FDE application, the line:<br>Set F77Lib\$Dir xxxx<br>(where 'xxxx' is the full path name of the directory where the Fortran libraries are kept.                |
| <i>link: (fatal) File name &lt;F77Lib\$Dir&gt;.fortlib not found</i> | there is not a file called 'Fortlib' in the Fortran libraries directory (see below)   | copy a new version from libf77.Fortlib from the original FDE zip file   |
| <i>File '... .. f77' not found</i>                                   | the Fortran compiler, 'f77', is not in the directory: <F77cl\$Dir>  | press F12 to leave the desktop and then type:<br>show F77cl\$Dir<br>which will show the directory where the f77 compiler should be. Copy f77 from the DDE   |
| <i>File '... .. link' not found</i>                                  | the linker (link) is not in the directory: <F77cl\$Dir>   | press F12 to leave the desktop type:<br>show F77cl\$Dir<br>which will show the directory where the linker should be. Copy 'link' from the DDE   |





# Getting Started

## Fortran Programming

Fortran is a compiled programming language.

This means that the program is first **written** in plain ASCII text in a Fortran source file, probably using a text editor like !SrcEdit. The source is then **compiled** into instructions the RISC OS hardware understands and stored in an object file. This file is **linked** with libraries of similar objects to make an executable image file which can finally be **run**.

### Tutorial

This section shows how to use the Fortran Development Environment (FDE) with a very simple example, and then expands it to show how to use some of the other options. Details of all options are in the [Reference Section](#).

First create a directory for your Fortran project, with an 'f' (Fortran source) sub-directory below it. Any source files for this project must be stored in this directory so that the compiler can find them. The FDE will create similar sub-directories as it needs them for holding object files, compiler listings, error messages etc. Three examples used in this chapter are included in the f directory in the FDE distribution zip file.

### Writing a program


Now type in the following program using the !SrcEdit you have just set up and save it with the name 'Hello' into the 'f' directory. (!SrcEdit works in a very similar way to !Edit but has extra features for Fortran programming shown below).

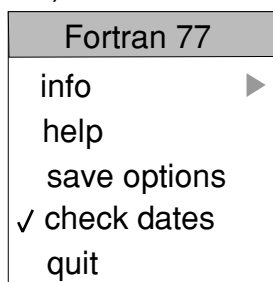
```
PROGRAM HELLO
PRINT *, 'Hello Fortran world.'
END
```

Remember each line must start with at least 6 blanks and have <Return> at the end (especially the last line – the caret should end up at the beginning of the line after the END).

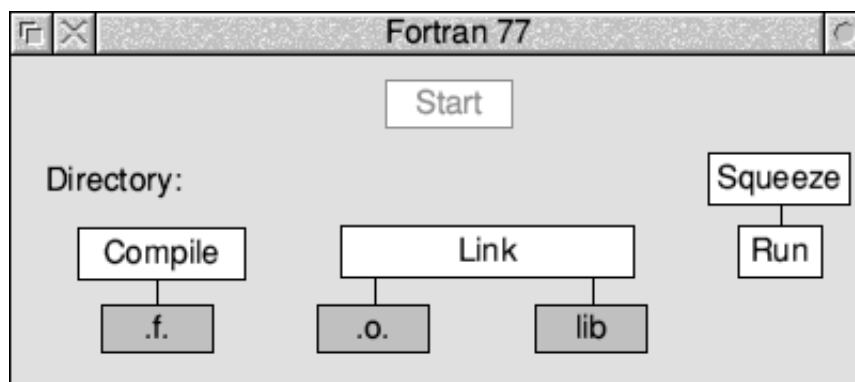
Now you can try out the on-line help for keywords: select the word PRINT and press <F1> which should show what this function does.

### Fortran Desktop Environment

Double click on the !FDE icon, , in the filer window to load the Fortran Development Environment (FDE) application on to the icon bar. Click <Menu> (the middle mouse button) over this icon to display this menu:



Most of the items in this menu have obvious meanings: moving the pointer over the arrow for "info" or clicking on "help" show windows with the FDE version and enough help to get started. If there is a tick by "check dates", remove it by clicking over its entry. This useful feature is described later. Now click <Select> on the FDE icon to open the development environment window shown below:



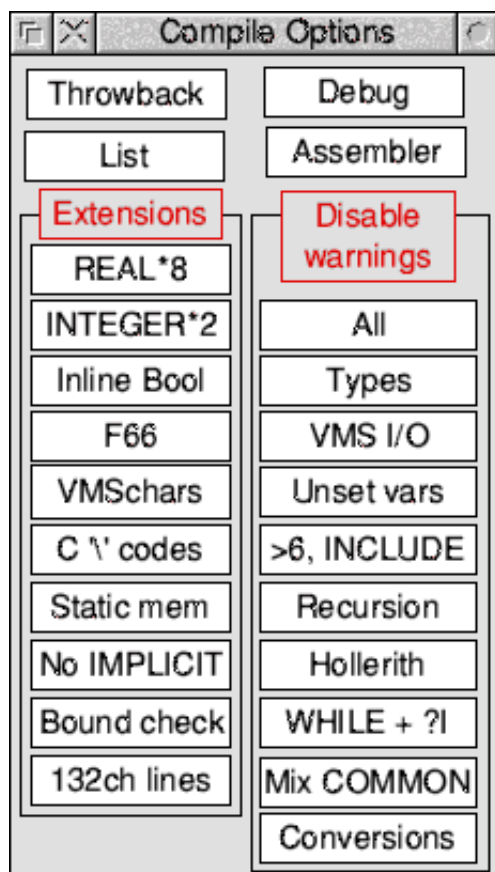
Drag your file 'Hello' into this window, and you will see it listed in a yellow icon connected to the 'Compile' box and the path name of your project directory shown above the white operations boxes.

### !Help

To get an interactive explanation of the functions of the icons in this window, and how the mouse interacts with them, use the interactive help application which is in your Apps directory. Read your RISC OS User Guide to find out how to use !Help.

### Compiler options

Before compiling, check which options are turned on by clicking <Menu> over the 'Compile' box. This will display a window of frequently used options. Their meanings are explained on the next page.



Turn off any selected options, (white on black background) by clicking over them with <Adjust>.

In options windows, <Select> selects the option, while <Adjust> deselects it; *they are not toggled*.

| Compiler Options                           | meaning   |
|--|---|
| Throwback                                  | produces a 'throwback' window of errors in the source code; clicking on an error leads straight to the faulty line                  |
| List                                       | makes a listing of the source with line numbers in a new directory 'l'  |
| Debug                                      | inserts debug code into the object file (and the executable image after linking)  |
| Assembler                                  | creates an assembler listing in a directory 's' rather than object code for linking   |
| <b>Optional extensions to the compiler</b> |   |
| REAL*8                                     | Treat all REAL*4 variables as REAL*8  |
| INTEGER*2                                  | Treat all integers as INTEGER*2   |
| Inline BOOL                                | converts functions like: ISHFT, IAND, IBCLR to in-line code (otherwise they will reference the Fortran library)                     |
| F66  | allow some Fortran66 constructs   |
| VMSchars                                   | allows and interprets the VMS characters: ?_!<>%&<br>Most useful of these is '!' which can be followed on a source line by comments |
| C '\ ' codes                               | Interpret '\ ' like a UNIX escape sequence:<br>e.g. '\n' = <newline> etc.   |
| Static mem                                 | forces all temporary variables into fixed locations (not on the stack)  |
| No IMPLICIT                                | forces IMPLICIT NONE so that all variables must be typed (INTEGER, REAL etc.)   |
| Bound Check                                | Inserts code to check for using indices outside the dimensions of arrays  |
| 132ch Lines                                | Treats characters 1-132 on each line as significant rather than the standard 1-72   |
| <b>Disable Warnings</b>                    |   |
| All  | Ignores all compiler warnings   |
| Types                                      | suppresses warnings for types BYTE, DOUBLECOMPLEX etc.  |
| VMS I/O                                    | suppresses warnings for VMS I/O statements like ACCEPT,DECODE,TYPE.   |
| Unset vars                                 | do not complain about unset variables   |
| >6, INCLUDE                                | allow various input relaxations, e.g. long names, tabs, "" string delimiters etc.   |
| Recursion                                  | suppress warning on direct recursive call   |
| Hollerith                                  | suppress warning when Hollerith initialises CHARACTER and friends   |
| WHILE +?l                                  | suppress warning for ?lc004, DO WHILE etc.  |
| Mix COMMON                                 | suppress warning for mixed variable types in COMMON   |
| Conversions                                | allow non-ANSI conversions, e.g. LOGICAL=string, INTEGER=LOGICAL etc.   |

## **Pragmas**

A pragma is an in-line command to the compiler notably for changing options. They appear with `#pragma` at the beginning of a line. A list of available pragma directives can be found in the main Fortran manual in Part 2 Section 3. Here we just show an example piece of code:

```

PROGRAM TBSLSH
C      to test the compilation of the backslash character: '\ '
C
#pragma c_style_esc
      CALL PRT('ONE\0')
      PRINT *, ' should be "ONE", followed by null '
      PRINT *
#pragma no_c_style_esc
      CALL PRT('\ char')
      PRINT *, 'should be "\ char" '
      STOP
      END
C
      SUBROUTINE PRT(STRING)
      CHARACTER *(*) STRING
      CHARACTER *1 C
      DO 10 I=1,LEN(STRING)
        C = STRING(I:I)
        PRINT 101,C,ICHAR(C)
101   FORMAT(1X,A1,1X,Z2)
10   CONTINUE
      RETURN
      END

```

The pragma `c_style_esc` is normally set in the compiler options as “C ‘\’ codes” in the table above. In this unlikely example it was needed to treat the backslash character as a C-type escape code to insert a null value into a string and also to have it interpreted with its normal ASCII value of 5C(hex).

## **Compiling**

In the main window, click over the ‘Compile’ box which turns black with white lettering while the ‘Start’ in the box at the top turns from grey to black with a cream background. Start compilation by clicking on this box which will turn blue, and contain the phrase ‘Checking Files’ while it makes sure it can find all the files you are going to need and they have the correct protection (read/write/locked etc.). Then the phrase changes to ‘Compiling Hello’, returning to its black ‘Start’ condition when it has finished. If you have a hard disc and all this happened too quickly to follow, click on ‘Start’ again to repeat the process.

## **Errors in compilation**

Although this is a short program, you may make a typing mistake so that the program does not compile. Try putting a deliberate mistake into your program and compiling again; e.g.

```

PROGRAM HELLO
PRNT *, 'Hello Fortran world.'
END

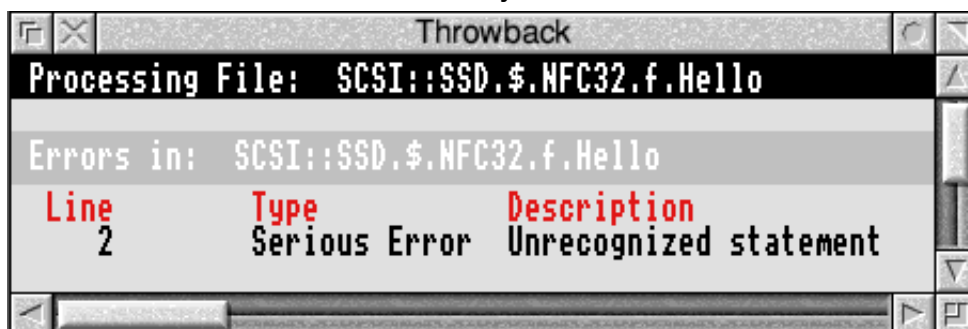
```

The statement `PRINT` is mistyped as `PRNT`. Now try to compile again, but instead of everything finishing normally, you get a standard RISC OS error window in the middle of the screen saying "Compilation Errors". Click over the 'Cancel' in this window, and a new window will appear with compiler error messages, much like this:

Norcroft RISC OS ARM Fortran vsn 5.80 (5.80.1.6) [30 Jan 2020]  
 "f.hello", line 2: Serious error: Unrecognized statement  
 f.hello: 0 warnings, 0 errors, 1 serious error

### Throwback

Now try out the 'Throwback' method of correcting compilation errors. Click with <select> over 'Throwback' in the Compile options window and try compiling again. This time, instead of the RISC OS error window, you see:



– a throwback window, which is another way of displaying your compilation errors. If you have a lot of errors in a long program, throwback helps you to make corrections more quickly. Throwback is completely described in the !SrcEdit documentation, but for now, just double click on the '2' below the red word 'Line'. This immediately brings up an SrcEdit window of your source file, open at the place where the error occurred, with the error line highlighted. Change `PRNT` to `PRINT`. Save your 'Hello' source, close the throwback window and recompile your program to make sure that it is now correct.

### On-line Fortran help

Now try the on-line help feature for Fortran in SrcEdit but first you have to tell !SrcEdit that you want help on Fortran. Click <Menu> over the SrcEdit icon on the icon bar and follow the menu tree: Options->Language and click on Fortran. Highlight the word `PRINT` in your source file by holding down <Select> while moving the pointer over the word. Press <F1> to see a new window showing an explanation of the statement `PRINT`. This on-line help knows about all the statements of the Fortran language. Be sure you select all the word and that it is correctly spelt. (See the [Application Libraries](#) chapter to find out how to access help for the applications libraries).

You should now have compiled 'Hello', and have its name in a yellow icon attached to the '.o.' box.

### Linking

First check that no link options are set by clicking <Menu> over 'Link' to bring up the link options window, click <Adjust> to deselect any highlighted options. Next click <Adjust> over 'Compile' in the main window to remove its highlight, and click <Select> over 'Link' to prepare for linking and then on 'Start'.

### Running a program

You should now have an executable image of 'Hello' in your base directory; the easiest way to run it is to double-click over its icon.

### **Compiling + linking + run**

The operation boxes 'Compile', 'Link' and 'Run' can all be selected at the same time in the main window. Clicking on 'Start' then runs them in sequence displaying the message from your program at the end.

### **Squeeze**

Squeeze is a utility which compresses executable image files. Compressed files can be run in the same way as ordinary files; they may be much smaller, often the decompressing time will be more than offset by the shorter time needed to read such a file from a slow disc. You can squeeze an executable image file either in a sequence like Compile-Link-Squeeze-Run, or by itself if you forgot to do it previously by clicking on its icon in the main window.

### **Libraries**

Now try an example which needs one of the supplied applications libraries. Type in another short program to make a beep noise:

```
PROGRAM SBEEP
C      sound a beep
      CALL BEEP
      PRINT *, 'beep sounded'
END
```

Save it in a new file 'f.sbeep'. If you still have the main FDE window with 'Hello' files in it, erase them by clicking <Adjust> over all 'Hello's, and then drop in your new source file. Select 'Compile', 'Link', 'Run' and then 'Start'.

### **Link errors**

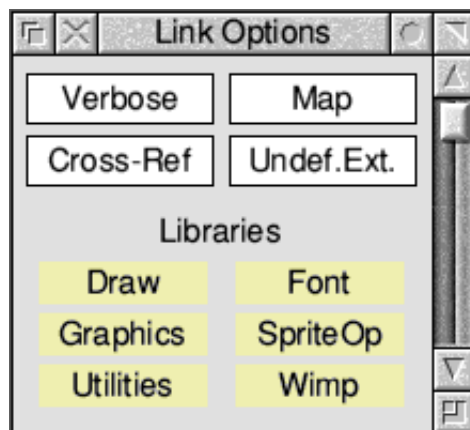
When it tries to link you will get a standard RISC OS error window in the middle of the screen saying "Link Error". Click on 'OK' to see an error message window showing something like this:

```
ARM linker: (error) Undefined symbol(s).
ARM linker:  beep_, referred to from o.SBeep
ARM linker: Errors in link, no output generated.
ARM linker: finished, 2 informational, 0 warning and 1 error message
```

The second line tells you that the routine 'BEEP' is missing. (The word is in lower case followed by an underscore because the Fortran compiler refers to its routines this way. See [Fortran Specific bindings](#) in the Reference Chapter for a more complete explanation of the way Fortran calls routines). BEEP is part of the 'Utilities' library; unlike the fortlb library, 'Utilities' is not automatically searched by the linker.

### **Accessing applications libraries**

Click <Menu> over 'Link' to display the Link Options window, and then <Select> the 'Utilities' library icon to highlight it, and add it to the list under the 'lib' box in the main window, which shows the libraries to be searched by the linker.



Now try your program again (you may leave out the compile step because the source is already compiled) and you should hear your program beep and print out the confirmation message.

### ***Removing libraries***

You can remove libraries from the linker list by clicking <Adjust> over their names either in the main or link options window.

### ***Application library test programs***

Each application library has the Fortran source of an associated test program, (e.g. Tutils for the Utilities library), in directory 'Tlibraries.f' in the zip file. These examples show how to call all the routines in the libraries. They contain many comments in the code and have on-screen instructions explaining how to use them. Copy Tutils into your base directory and try compiling, linking and running it. The file: 'Tlibraries.00Contents' has a summary of which parts of the Wimp library are tested.

### ***Run options***

You can add arguments to the run command line which are passed to the executing program, so you could pass file names or other useful information to a program at the time it is run. Try this example program:

```

PROGRAM COMMND
C   to read and decode a command line argument
      CHARACTER*20 LINE,ARG
C   get the command line which will be
C   'commnd myfile'
      CALL GTARGS(LINE)
C   get second word (which starts after column 7)
      JS = 7
      CALL NEXTWD(LINE,JS,ARG,LENGTH)
C   second word is in ARG, print it
      PRINT *, 'argument is:', ARG(1:LENGTH)
END

```

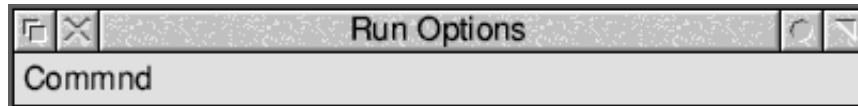
Save this file in 'f.commnd', then compile, link and run it with the 'Utilities' library. It should give the answer: *argument is:*

because we have not given it an argument. Deselect the 'Compile' and 'Link' from the main window.



### Run option window

Click with <Menu> over the 'Run' box (which should still be selected) and the Run Options window will appear; it contains the line which will be sent to the Command Line Interpreter.



Click over it to select it and type in <Space>myfile pressing <Return> at the end to remove this window. Now run the program again by clicking on 'Start'. This time the program should reply with: *argument is:myfile*.

### Running programs with options

This is the simplest way to add arguments to a command line from the desktop. Other ways are to open a RISC OS task window or leave the desktop altogether and enter the command line environment. Using Fortran from the [command line](#) is described in the Reference Chapter.

### Working directory structure

Now look at the filer window for your project directory. As well as the 'f' sub-directory of Fortran source files, there is now an 'o' sub-directory containing the compiled object files with the same names as the source files. The executable image files are in the project directory itself. Other sub-directories which the FDE may create are:

- 'e' containing error messages and linker listing files
- 's' containing assembler listings created with the 'Assembler' compiler option
- 'l' containing source listings using the 'List' compiler option

### Linker listings

Verbose, map and cross-reference listings from the linker are sent to the standard output file where errors are reported, so they appear as 'e.' files.

### Dragging files to the FDE

Fortran source files from an 'f' directory, compiled object files from an 'o' directory or executable image files from the same base directory can be dragged to the main window of the FDE. The directories 'f' and 'o' themselves may also be dropped when the entire contents will be used. No other file types are accepted. Dragging in files from a different base directory will remove all old file entries and change the 'Directory:' entry before loading the file. Files can also be dragged to the icon on the icon bar.

### Removing and reordering FDE files

File names can be removed from the lists in the main window simply by clicking <Adjust> over them. When you have more than one in a list, clicking with <Select> over an item moves it to the top of the list so you can change the order in which files are compiled or linked. This is very important in the library list because library routines which reference routines in another library must come before them in the list. The Fortran system library (Fortlib) is always searched last, so that it does not appear in the list.

***'Check dates'***

If you select the 'check dates' entry in the main menu (opened with <Menu> over the FDE icon) the time and date of Fortran source files you are going to use are compared with those of the corresponding object files.

If you try to compile a program for which there is a more recent object file, the compilation will be skipped (although FDE goes through the same sequence).

Recompile your 'Hello' program with this option selected (ticked) and you will find that the compilation step is skipped.

Alternatively, if you drag an object file onto the FDE icon or the main window for which the Fortran source is more recent, the entry will appear under the '.f.' column rather than the '.o.' column because it needs to be recompiled.

# **Fortran System Library**

## **Introduction**

This chapter describes the intrinsic functions provided by the Fortran compiler including all the ones defined in the ANSI specification and some extensions for the RISC OS computers. Most of the library is devoted to Input/Output with small parts for arithmetic and trigonometric functions, initialization termination and error handling. The library intrinsic to the FDE is described. It serves the same function as the standard one distributed by ([ROOL](#)) but has entries to serve some functions of the application libraries, e.g. to close special files on an abnormal exit. Four fast sorting routines (which used to be in the Utilities application library) are also included.

## **Documentation available**

The contents of the library are summarised below; (1) denotes extensions to the standard, (2) are generic function names, . The full details are available only in ASCII text files in the help provided by !SrcEdit: in the Iconbar menu of !SrcEdit follow Options->Language and select the library 'Fortran'; then select the desired name while it is displayed by !SrcEdit and press <F1>. The help files may be printed from !SrcEdit.help.HelpF77 ,

## **Routines by topic**

All the standard functions and extensions are listed here; most are implemented by the compiler.

### **Bit handling<sup>1</sup>**

BTEST IBCLR IBITS IBSET MVBITS<sup>2</sup>

### **Character**

CHAR ICHAR INDEX LEN LGE LGT LLE LLT LNBLNK<sup>1</sup>

### **Complex**

AIMAG CABS CCOS CEXP CLOG CMPLX<sup>2</sup> CONJG<sup>2</sup> CSIN CSQRT

### **Complex\*16<sup>1</sup>**

CDABS CDCOS CDEXP CDLOG CDSIN CDSQRT DCMPLX DCONJG

### **File Handling<sup>1</sup>**

FFSIZE FLGETP FLSETP FLSKIP

### **Hyperbolic**

COSH<sup>2</sup> DCOSH DSINH DTANH SINH<sup>2</sup> TANH<sup>2</sup>

### **Inverse hyperbolic<sup>1</sup>**

ACOSH ASINH ATANH DACOSH DASINH DATANH

### **Calling SWIs<sup>1</sup>**

KERNEL\_LAST\_OSERROR KERNEL\_SWI KERNEL\_SWI\_C

A simpler method for these is using ARCSWI in the [Utilities](#) library

### **Logical and shifting<sup>1</sup>**

IAND IEOR IOR ISHFT ISHFTC NOT

### **Logarithms**

ALOG ALOG10 DLOG DLOG10 LOG<sup>2</sup> LOG10<sup>2</sup>

**Numerical**

ABS<sup>2</sup> AINT<sup>2</sup> AMAX0 AMAX1 AMIN0 AMIN1 AMOD ANINT<sup>2</sup> DABS DDIM DEXP DIM<sup>2</sup>  
 DIMAG DINT DMAX1 DMIN1 DMOD DNINT DPROD DREAL DSIGN DSQRT EXP<sup>2</sup>  
 IABS IDIM ISIGN MAX<sup>2</sup> MAX0 MAX1 MIN<sup>2</sup> MIN0 MIN1 MOD<sup>2</sup> RND01 SETRND SIGN<sup>2</sup>  
 SQRT<sup>2</sup>

**System operations<sup>1</sup>**

LOC LOCC

**Date/time<sup>1</sup>**

CPU CDATE

**Trigonometric**

ACOS<sup>2</sup> ASIN<sup>2</sup> ATAN<sup>2</sup> ATAN2<sup>2</sup> COS<sup>2</sup> DACOS DASIN DATAN DATAN2 DCOS DSIN  
 DTAN SIN<sup>2</sup> TAN<sup>2</sup>

**Type conversion**

FLOAT IFIX INT<sup>2</sup> NINT<sup>2</sup> IDINT IDNINT DBLE<sup>2</sup> REAL<sup>2</sup> SNGL

**Sorting routines<sup>1</sup>**

QSORTC QSORTD QSORTI QSORTR

**Calling Sequences**

Only the functions which are extensions to the Fortran standard are listed here.

Documentation on all the routines is in the SrcEdit help file. Functions have an I, L, R, D or C in the first column below, for integer, logical, real, double precision or complex\*16 functions, otherwise routines are subroutines.

|   |                 |  |
|---|-----------------|--|
| R | ACOSH(R)        | REAL*4 inverse hyperbolic cosine                   |
| R | ASINH(R)        | REAL*4 inverse hyperbolic sine                     |
| R | ATANH(R)        | REAL*4 inverse hyperbolic tangent                  |
| L | BTEST(I,J)      | .TRUE. if Jth bit of I is on                       |
| D | CDABS(C)        | absolute value of COMPLEX*16 argument              |
|   | CDATE(STR)      | gets current date and time                         |
| C | CDCOS(C)        | COMPLEX*16 cosine                                  |
| C | CDEXP(C)        | COMPLEX*16 exponential                             |
| C | CDLOG(C)        | COMPLEX*16 natural logarithm                       |
| C | CDSIN(C)        | COMPLEX*16 sine                                    |
| C | CDSQRT(C)       | COMPLEX*16 square root                             |
| R | CPU(TIME)       | CPU time used since last called                    |
| D | DACOSH(D)       | REAL*8 inverse hyperbolic cosine                   |
| D | DASINH(D)       | REAL*8 inverse hyperbolic sine                     |
| D | DATANH(D)       | REAL*8 inverse hyperbolic tangent                  |
| C | DCMPLX(X[,Y])   | converts one or two REAL*8 arguments to COMPLEX*16 |
| C | DCONJG(C)       | COMPLEX*16 complex conjugate                       |
| D | DIMAG(C)        | REAL*8 imaginary part of COMPLEX*16                |
| D | DREAL(C)        | REAL*8 real part of COMPLEX*16                     |
|   | FFSIZE(U,N,J,R) | finds size of a direct access file                 |
|   | FLGETP(U,I)     | finds current position in a file                   |
|   | FLSETP(U,I)     | sets current position in a file                    |
|   | FLSKIP(U,IER)   | skip to the end of file                            |
| I | IAND(I,J)       | Boolean AND of I and J                             |
| I | IBCLR(I,J)      | clears the J <sup>th</sup> bit of I                |

- I IBITS(I,J,K)      extracts K bits starting at bit J from I
- I IBSET(I,J)        sets the J<sup>th</sup> bit of I
- I IEOR(I,J)        Boolean exclusive OR of I and J
- I IOR(I,J)         Boolean OR of I and J
- I ISHFT(I,J)        left shift I by J bits, (-J for right shift)
- I ISHFTC(I,J,K)    returns I with I.s. K bits rotated left by J
- KERNEL\_LAST\_OSERROR(N,M) to find a Software Interrupt (SWI) error
- L KERNEL\_SWI(N,I,O) to cause an SWI
- L KERNEL\_SWI\_C(N,I,O) to cause an SWI returning a C flag
- I LNBLNK(STR)      returns the position of last non blank in string STR
- I LOC (J)          returns address of word (any non character type)
- I LOCC(STR)        returns address of a string of type character
- MVBITS(M,I,L,N,J) moves L bits at I in M to bit J in N
- I NOT(I)            Boolean NOT of I
- QSORTC(C,INDX,N)    sorts pointers to a CHARACTER array
- QSORTD(D,INDX,N)    sorts pointers to a REAL\*8 array
- QSORTI(IN,INDX,N)   sorts pointers to an INTEGER array
- QSORTR(R,INDX,N)    sorts pointers to a REAL array
- R RND01()          returns a pseudo-random number in range 0 to 1
- SETBUF(I,N)        sets the buffer size for unit I to N
- SETRND(I)         initialises the pseudo-random number sequence

***I/O extensions***

The many extensions to the standard Fortran77 input/output instructions can be found in the [Input / Output Extensions](#) section of the reference section.

# **Application Libraries**

### ***Introduction***

This chapter describes libraries of routines which provide simple interfaces to the RISC OS operating system and to the special hardware of these 32-bit computers. These relocatable libraries are already in the directory !FDE.libf77 and the sources are in source.libraries.s

### ***Available Libraries***

The distributed libraries are:

- [Draw](#) file creation
- [Font](#) handling
- [Graphics](#)
- [Sprite](#) operations
- [Utilities](#) mostly for referencing RISC OS
- [Wimp](#) environment

Each library has a corresponding test program which illustrates calls to all of its routines. The test source files are in Tlibraries.f

There is a [tutorial](#) in this manual on how to write a program to run in the Wimp together with 15 example programs.

### ***Naming conventions***

Routine parameters follow the Fortran conventions for implicit naming of variables; names beginning with letters I to N are INTEGER\*4, all others are REAL\*4, unless specified by an explicit type statement. Many of the libraries have routines specifying terminal screen positions in the horizontal, x, and vertical, y, directions. These usually have the variable names like IX,IY, e.g. CALL GRMOVE(IX,IY).

### ***Documentation available***

The contents of all libraries are summarised in this chapter. Full details are available in ASCII text files in the help provided by !SrcEdit ([see installation section](#)). The description of each library member contains:

- The format of the calling sequence
- The purpose of the routine
- Definitions of the type and contents of the arguments: (these can be assumed to be input to the sub-program unless the definition explicitly states otherwise; e.g. "returns...")
- The result (for functions)
- Any errors which may occur
- Comments



## **Draw library**

This library contains a collection of routines to make Draw files from text and vector graphics. These can then be viewed with the !Draw utility and thence printed..

### **Naming Conventions**

The procedures in this library have names beginning with the prefix 'DW' followed by four characters indicating the function of the routine.

### **Creating a Draw File**

This is done by opening the file, creating one or more 'objects' and then closing the file. Colours, line styles, and fonts are set up separately in utility routines. (Note that fonts must be defined before any objects are drawn).

### **Example**

```
C      set up circle radius 0.4, centre (0.5, 0.5)
      DIMENSION XYC(2)
      DATA XYC/0.5,0.5/
C      initialise/open the draw file 'MYDRAW'
C      define user area for A4 portrait
      CALL DWINIT('MYDRAW','A4P',0.,0.,1.,1.4,IERR)
C      plot circle radius 0.4 (outline only)
C      with default colour, line thickness etc.
      CALL DWCIRC(XYC, 0.4, IERR)
C      close and save the file
      CALL DWDONE(.TRUE.)
```

### **Co-ordinate Systems**

The user system is fairly arbitrary, except that it has to define a positive area. It is advisable to use an aspect ratio not very different from a standard A4 type page ( $1:\sqrt{2}$ ) in order to take advantage of the available area.

Font sizes are measured in printers' points (1/72 inches); line-widths and dot/dash sizes are measured in OS units.

### **Page sizes**

These are the standard paper sizes, from A5 to A0, portrait or landscape; they are defined by a character string, like 'A4P' or 'A5L' to the initialisation routine.

### **Initialisation routine**

DWINIT

### **Creating objects routines**

DWARCC DWCIRC DWCURV DWELIP DWJPEG DWJPSZ DWPOLY DWSECT  
DWSEGC DWSVRT DWTEXT DWTXTA

### **Changing colours routines**

DWCOLF DWCOLL DWTXBC DWTXFC

### **Changing line styles routines**

DWDASH DWLWID DWSTYL

### **Changing text style routines**

DWFONT DWSZXY DWTXFS

### **Grouping objects routines**

DWBGRP DWEGRP

### **Closing the draw file routine**

DWDONE

**Errors**

The following errors can occur while creating a Draw file (no file is created for positive error numbers):

- 1 can not open file
- 2 file has not been opened
- 3 unrecognised page size
- 4 user's dimensions do not define a positive area
- 1 object outside user area
- 2 unknown font
- 3 illegal colour
- 4 can't write any more to file (disc full?)
- 5 file already open
- 6 less than 2 points on path
- 7 group already started
- 8 no group to end
- 9 no more room to store font name
- 10 too late to store fonts
- 11 bad font handle
- 12 text size outside range
- 13 illegal angles for arc
- 14 identical consecutive points on curve
- 15 can't find sprite

If the Draw file is not closed, the system will close and keep the file when the program exits.

**Calling Sequences**

|  |   |
|--|---|
| DWARCC (XYC,R,IH1,IH2,IERR)            | draws a circular arc  |
| DWBGRP(IERR)                           | starts a group  |
| DWCIRC (XYC,R,IERR)                    | draws a circle  |
| DWCOLF(IR,IG,IB)                       | sets the fill colour  |
| DWCOLL(IR,IG,IB)                       | sets the line colour  |
| DWCURV( XY,N,CLOSE,IERR)               | draws a curve   |
| DWDASH(N,L1,L2,etc..)                  | sets dashed lines format                                      |
| DWDONE(DELETE)                         | closes the draw file  |
| DWEGRP(IERR)                           | ends the group  |
| DWELIP(XYC,A,B,THETA,IERR)             | draws an ellipse  |
| DWFONT(FONTNM,IHF)                     | sets up a font  |
| DWINIT(FILENM,PAGESZ,XL,YL,XH,YH,IERR) | initialises the draw file                                     |
| DWJPEG(JPEG,L,XY,IERR)                 | Plots JPEG image from array<br>JPEG, length L, at position XY |
| DWJPSZ(JPEG,L,XY,IERR)                 | finds the size (XY) of a JPEG<br>image in user co-ordinates   |
| DWLWID(IWID)                           | sets the line width   |
| DWPOLY(XY,N,CLOSE,IERR)                | draws polyline or polygon                                     |
| DWSECT(XYC,R,IH1,IH2,IERR)             | draws a sector of a circle                                    |
| DWSEGC(XYC,R,IH1,IH2,IERR)             | draws a segment of a circle                                   |
| DWSPRT(IA,SPN,XY,IERR)                 | inserts a sprite  |
| DWSTYL(J,IEC,IBC,IWR,ICW,ICL)          | sets the line style   |
| DWSZXY(String,XY,IERR)                 | finds size of text image                                      |
| DWTEXT(XY1,String,XY2,IERR)            | draws one line of text  |
| DWTXBC(IR,IG,IB)                       | sets the font background colour                               |
| DWTXFC(IR,IG,IB)                       | sets the text foreground colour                               |
| DWTXFS(IHF,IXS,IYS,IERR)               | sets up current text font and size                            |
| DWTXTA(XY,ANG,STR,SZ,IERR)             | draws text at any angle                                       |



## **Font Library**

The routines in this library let you display character strings on the VDU using outline fonts. You first select a font by its name and size and then send lines of text to the screen either justified or unjustified. The minimum you need to display text in an outline font is:

- CALL FTFIND to set up the font
- CALL FTPRNT to display the text
- CALL FTLOSE to release the font

### **Routines in the library**

All routines are subroutines having names beginning with 'FT' They have their input parameters ordered before their result parameters.

### **Co-ordinates**

The co-ordinates and dimensions used are almost always measured in standard OS units. The standard printer point size (1/72 inch) is nominally 2.5 OS units. Milli-points (400 milli-points = 1 OS unit) are only used in control sequences embedded in a display string so that the following text can be displaced, e.g. for sub- and superscripts.

### **Finding fonts routines**

FTFIND FTLIST FTLOSE FTPREF

### **Display routines**

FTPRNT FTPRTA

### **Finding sizes routines**

FTBBOX FTSIZC FTSZXY

### **Colours routines**

FTGETC FTGETP FTGETT FTSETC FTSETP FTSETT

### **Font cache routines**

FTCACH FTRFMX FTWFMX

### **Caret routines**

FTFCRT FTSCRT

**Calling Sequences**

|                                   |   |
|-----------------------------------|---|
| FTBBOX(IH,IBX,IBY)                | gets font bounding box                  |
| FTCACH(ISZ,IU)                    | gets font cache size                    |
| FTFCRT(IH,S,IXIN,JU,IX,NPC,INDX)  | finds caret position                    |
| FTFIND(NM,IY,IX,IH)               | finds a font in the font store          |
| FTGETC(IH,IF,IB,IOFF)             | gets colours of font                    |
| FTGETP(IH,IBK,IFG,IOFF)           | sets colour palette                     |
| FTGETT(ITHR)                      | gets anti-alias thresholds              |
| FTLIST(NAME)                      | gets available font names               |
| FTLOSE(IH)                        | loses a font                            |
| FTPREF(IH,PFX)                    | finds directory prefix                  |
| FTPRNT(IH,S,IX,IY,BL,JU)          | displays a character string             |
| FTPRTA(IH,S,IX,IY,BL,A,RMAG)      | displays a string at any angle and size |
| FTRFMX(IFMX,MX15)                 | reads maximum size of cache             |
| FTSCRT(IH,IC,IX,IY)               | sets up caret                           |
| FTSETC(IH,IB,IF,IO)               | sets up logical colours                 |
| FTSETP(IH,WP,LB,LF,IB,IF)         | sets up colour palette                  |
| FTSETT(ITHR)                      | sets anti-alias thresholds              |
| FTSZC(IH,ICH,IBX,IBY)             | gets character size                     |
| FTSZXY(IH,S,IXMN,IYMN,IXMX,IYMX,) | gets string bounding box                |
| FTWFMX(IFMX,MX15)                 | sets maximum size of cache              |

## **Graphics Library**

This library contains subroutines for low level graphics.

### **Naming Conventions**

Their subroutine names begin with the prefix 'GR', followed by a maximum of 4 letters making a name similar to the BASIC graphics commands.

### **Co-ordinates Units and Origin**

The routines use graphics screen co-ordinates in OS screen units (unless stated otherwise), with the origin at the lower left of the screen; for old screen modes the top right corner extends to 1279 horizontally, x, and 1023 vertically, y. They should have values between -32768 and 32767, and be variables of type INTEGER\*4. (Using INTEGER\*2 variables for these graphics routines will give unpredictable results.) Using OS units for graphics co-ordinates makes the code mode independent.

Text co-ordinates are given as numbers of characters and rows. The origin is at the top left of the screen, y increases downwards. The maxima depend on the screen mode.

### **Pixel sizes**

The screen image is actually composed of pixels; for normal modes 4 OS units correspond to one pixel; in multisync modes, 2 OS units usually correspond to a pixel. In some modes, the scaling is different in the x and y directions. Routine GRRMV can return the number of OS units in a pixel.

### **Windows**

In these routines the term 'Window' means an area of the screen where text or graphics may be displayed, sometimes known as a 'View Port'; it does not mean an area of screen which can be moved around on the RISC OS Desktop using the mouse.

### **Text routines**

GRCHAR GRCURS GRSETT GRTAB GRWBIG GRWOG GRWOT

### **Whole screen routines**

GRCLRG GRCLRT GRSETM GRORIG GRWAIT

### **Window routines**

GRGWIN GRTWIN

### **Colour routines**

GRGCOL GRGETC GRSETC GRTCOL  
(palette)  
GRGETL GRPAL

### **Drawing routines**

GRARCC GRCIRC GRCOPY GRDEFD GRDPOL GRDRAW GRDTRI GRELIP  
GRFILB GRFILF GRLINE GRMOVE GRPLOT GRPOLY GRRECT GRSECT  
GRSEGC GRSPOT GRTRI GRVDU

### **Routines for determining variables**

GRCMV GRGETM GRRMV GRRVV

### **Mouse/pointer operations routines**

GRHOUR GRPBOX GRPBUT GRPGET GRPPUT GRPSET GRPSPD

**Calling Sequences**

Those routines marked with a (\*) must not be used in a Wimp Program.

GRARCC(IX,IY,IR,A1,A2) draws arc of circle  
 GRCHAR(ICH,IDEF) character definition  
 GRCIRC(IX,IY,IR,FILL) draws outline or filled circle  
 GRCLRG\* clears the graphics window  
 GRCLRT\* clears the text window  
 GRCMV(M,N) checks valid screen mode  
 GRCOPY(IX1,IY1,IX2,IY2,IXN,IYN) copies a rectangle  
 GRCURS(ISTAT)\* turns the flashing cursor on/off  
 GRDEFD('1100') defines the dash pattern for line drawing  
 GRDPOL(IXY,N,C,F,IW,ID) draw a polygon  
 GRDRAW(IX,IY) draws line to point  
 GRDTRI(IEND,IWID,L) sets up triangular end-caps  
 GRELIP(IX,IY,MA,MI,A,FILL) draws an outline or filled ellipse  
 GRFILB(IX,IY) fills to a non-background colour boundary  
 GRFILF(IX,IY) fills to a foreground colour boundary  
 GRGCOL(IA,IC) sets graphics colour (2,4,16 colour modes)  
 changes extended colour fill patterns  
 GRGETC(IX,IY,L) finds colour of point (2,4,16 colour modes)  
 GRGETL(IX,IY,LC) finds the logical colour at IX,IY  
 GRGETM(M) gets the screen mode or mode selector  
 GRGWIN(IX1,IY1,IX2,IY2)\* makes a rectangular graphics window  
 GRHOUR(IHG) turns Hourglass pointer on/off  
 GRLINE(IX1,IY1,IX2,IY2) draws a line between 2 points  
 GRMOVE(IX,IY) moves to point  
 GRORIG(IX,IY)\* translates graphics origin to point  
 GRPAL(L,IR,IG,IB)\* sets colour for logical colour  
 (2,4 16 colour modes)  
 GRPBOX(IX1,IY1,IX2,IY2) sets mouse/pointer boundary  
 GRPBUT(IX,IY,IB)\* waits for mouse/keyboard interrupt  
 GRPGET(IX,IY,IB)\* gets pointer position and mouse button  
 GRPLOT(K,IX,IY) general plotting routine  
 GRPOLY(N,IX,IY,FILL) draws a polyline through N points  
 GRPPUT(IX,IY) positions mouse pointer  
 GRPSET(IS) turns on/off pointer  
 GRPSPD(ISPD) sets pointer/mouse speed ratio  
 GRRECT(IX1,IY1,IX2,IY2,FILL) draws outline or filled rectangle  
 GRRMV(M,N,I) Read Mode Variable  
 GRRVV(N,I) Read VDU Variable  
 GRSECT(IX,IY,IR,A1,A2,FILL) draws sector of circle  
 GRSEGC(IX,IY,IR,A1,A2,FILL) draws segment of circle  
 GRSETC(IA,IR,IG,IB) sets current graphics colour  
 GRSETM(M) set mode or mode selector  
 GRSETT(IR,IG,IB) sets text colour  
 GRSPOT(IX,IY) plots a single pixel  
 GRTAB(IX,IY)\* position for text printing in text window  
 GRTCOL(ICOL)\* sets text colour (2,4,16 colour modes)  
 GRTRI(IX1,IY1,IX2,IY2,IX3,IY3,FILL) draws outline or filled triangle  
 GRTWIN(IX1,IY1,IX2,IY2)\* sets a rectangular text window



## ***Graphics Library***

## **Application Libraries**

GRVDU(I)\*                sends one byte to the screen driver  
GRWAIT                waits for vertical synchronisation  
GRWBIG(IX,IY,TEXT,IS)\* prints enlarged characters  
GRWOG(IX,IY,TEXT)\*       prints graphics text  
GRWOT(IX,IY,TEXT)\* prints text at tab position in text window



## **Sprite Operations Library**

This library contains functions for manipulating Sprites.

### **Naming Conventions**

All routine names in this library begin with the prefix 'SP'

They emulate most of the OS\_SpriteOp commands (SWI &2E), though some are simplified. The operation used and the reference in the RISC OS 2 and 3 Programmer's Reference Manuals is given for each routine.

The routines are LOGICAL FUNCTIONS, which return .TRUE. if the operation has been completed correctly, or .FALSE. if an error has occurred. Details of an error can be found by calling subroutine SPERR.

### **Parameters**

All parameters are INTEGER\*4, except for sprite and file names which are CHARACTER, (e.g. 'SN' and 'FL' in this document), and scaling factors which are COMPLEX. Trailing blanks in the CHARACTER names are ignored, hence arrays of sprite names can be defined easily. The sprite names have a maximum length of 12 characters; characters beyond 12 are ignored.

The values of the parameters are all input to the functions unless it is specifically stated that the value is 'returned'.

### **Sprite storage areas**

Sprites are stored either in a user area, or the system area. The routines decide which one to use by looking at the contents of the first word of the sprite area, ISTORE, thus:

=0, system area

>0, user area. (Note, the first word of a user sprite area contains the length of the area in bytes which is always positive.)

### **Sprite names**

Sprites in any area are defined by their name (CHARACTER). Sprites in a user area may be accessed faster by using their address, (INTEGER), which replaces the CHARACTER name in the calling sequence.

Sprite addresses are found using the routine SPADDR.

### **Palettes and Pixel Translation**

These are ignored for screen modes with more than 256 colours.

### **Whole screen operations routines**

SPSLOD SPSSAV

### **Filing sprites routines**

SPALOD SPAMRG SPASAV

### **Pointer routine**

SPSETP

### **Sprites routines**

SPADD SPADDR SPASIZ SPASYS SPCOPY SPCPAL SPDEFC SPDEFR SPDELC  
SPDELP SPDELR SPDELS SPFLPX SPFLPY SPINFO SPINIT SPINSC SPINSR  
SPLEFT SPNAME SPPL SPPLAA SPPLSC SPPLTR SPPLXY SPRENA SPRESV  
SPRPIX SPWPIX

### **Sprite Mask routines**

SPCMASK SPDELM SPPM SPPMSC SPPMXY SPRMSK SPWMSK

**Character routines**

SPCHAR

**Switching the screen routines**

SP2MSK SP2SCR SP2SPR

**Error reporting routines**

SPERR

**Calling Sequences**

In these routines, ISR is the sprite area, SN the sprite name and FL the file name

|                                   |  |
|-----------------------------------|--|
| SP2MSK(ISR,SN)                    | switches VDU output to the mask of a selected sprite |
| SP2SCR                            | resets VDU output back to the screen                 |
| SP2SPR(ISR,SN)                    | switches VDU output to a selected sprite             |
| SPADD (ISR,S1,S2,IB)              | adds sprite 'S2' to 'S1' in area ISR                 |
| SPADDR(ISR,SN,IADR)               | finds address of a sprite in area ISR                |
| SPALOD(ISR,FL)                    | loads a sprite area from file                        |
| SPAMRG(ISR,FL)                    | merges the area from file to sprite area ISR         |
| SPASAV(ISR,FL)                    | saves a sprite area to file                          |
| SPASIZ(ISR,NS,ISZ,JF)             | gets characteristics of sprite area                  |
| SPASYS(NK)                        | changes size of system sprite area                   |
| SPCHAR(ICH,IX,IY,SC)              | plots scaled ASCII character ICH at IX,IY            |
| SPCMSK(ISR,SN)                    | creates a mask for a sprite                          |
| SPCOPY(ISR,SN,SN2)                | copies a sprite to its own area                      |
| SPCPAL(ISR,SN)                    | creates a palette for a sprite                       |
| SPDEFC(ISR,SN,IP)                 | defines a sprite from graphics cursors               |
| SPDEFR(ISR,SN,IP,IX1,IY1,IX2,IY2) | defines a sprite from user co-ordinates              |
| SPDELC(ISR,SN,ICLM)               | deletes a column from a sprite                       |
| SPDELM(ISR,SN)                    | deletes the mask from a sprite                       |
| SPDELP(ISR,SN)                    | deletes the palette from a sprite                    |
| SPDELR(ISR,SN,IROW)               | deletes a row from sprite                            |
| SPDELS(ISR,SN)                    | deletes a sprite                                     |
| SPERR (IERR,ERRTXT)               | returns an error number and message                  |
| SPFLPX(ISR,SN)                    | flips a sprite about its X-axis.                     |
| SPFLPY(ISR,SN)                    | flips a sprite about its Y-axis.                     |
| SPINFO(ISR,SN,IW,IH,MA,MO,LP)     | gets characteristics of a sprite                     |
| SPINIT(LEN,ISR)                   | initialises array ISR as a sprite area               |
| SPINSC(ISR,SN,ICLM)               | inserts a blank column into sprite                   |
| SPINSR(ISR,SN,IROW)               | inserts a blank row into sprite                      |
| SPLEFT(ISR,SN)                    | removes wastage from a sprite                        |
| SPNAME(ISR,ISP,SN)                | finds name of the ISP <sup>th</sup> sprite in ISR    |
| SPPL(ISR,SN,IA)                   | plots a sprite at the graphics cursor                |
| SPPLAA(ISR,SN,IX,IY,SC,IPX)       | plots scaled anti-aliased sprite (not in RISC OS>3)  |
| SPPLSC(ISR,SN,IX,IY,IACT,SC,IPX)  | plots scaled sprite                                  |
| SPPLTR(ISR,SN,IFL,IS,IA,TR,IPX)   | plots a transformed sprite (not RISC OS 2)           |
| SPPLXY(ISR,SN,IX,IY,IA)           | plots a sprite at user co-ordinates                  |
| SPPM(ISR,SN)                      | plots the mask of a sprite at graphics cursor        |
| SPPMSC(ISR,SN,IX,IY,SC)           | plots the scaled mask of a sprite                    |
| SPPMXY(ISR,SN,IX,IY)              | plots the mask of a sprite at user co-ordinates      |
| SPRENA(ISR,SN,SN2)                | renames a sprite                                     |

SPRESV(ISR,SN,IP,IW,IH,MO) reserves space for a sprite  
SPRMSK(ISR,SN,IX,IY,MASK) reads state of the mask at pixel (IX,IY)  
SPRPIX(ISR,SN,IX,IY,IC,IT) reads the colour of pixel at (IX,IY)  
SPSETP(ISR,SN,IB,IX,IY,SC,IPX) sets the pointer shape from sprite  
SPSLOD(FL) loads graphics screen from a file  
SPSSAV(FL,IPAL) saves graphics screen to file  
SPWMSK(ISR,SN,IX,IY,MASK) writes to a mask pixel  
SPWPIX(ISR,SN,IX,IY,IC,IT) writes colour to a sprite pixel



**Utilities Library**

This library contains routines for communicating with the operating system and sending information directly to a printer driver; other routines are included for backwards compatibility with Acorn Fortran77 release 2. Besides the test program 'Utilities' for most of the routines in this library, there is also a 'Tprinter' for testing the print routines.

**RISC OS Interface routines**

ARCSWI FPSSET FPSTAT GTARGS IGET INKEY MOUSE RSPACE TRACEP

**Filing System routines**

FLACCS FLDATE FLDIRE FLLIST FLLOAD FLSAVE FLSIZE FLTIME FLTYPE

**Messages routines**

MSGCLS MSGGET MSGGGS MSGOPN MSGSIZ

**Print commands**

PRCLOS PRINFO PRKILL PROPEN PRPBEG PRPEND PRSIZE

**Sounds routines**

BEEP SOUND VOICE

**Miscellaneous routines**

COPY2L COPY2U EVALF JVALK NEXTWD

**Fortran77 release 2 compatibility routines**

OSBYTE OSBYTE1 OSBYTE2 OSCLI OSGETERROR OSWORD

These are for communicating with the Operating System in the old 8-bit BBC-Micro format. Their meanings can be found in most RISC OS computer reference manuals.

**Sorting routines**

The Sorting routines: QSORTC QSORTD QSORTI QSORTR used to be in this library but have been moved to the system library to be compatible with the fortlb supplied with the compiler.

**Calling Sequences**

Routines are listed alphabetically by name. They are subroutines unless there is a character in first column below.

I = Integer Function, L = Logical Function, R = Real Function

|                           |  |
|---------------------------|--|
| ARCSWI(SWI,IREGS,IER,ERM) | causes a Software Interrupt            |
| BEEP                      | sounds the 'bell'                      |
| COPY2L(A,B)               | copy string to lower case              |
| COPY2U(A,B)               | copy string to upper case              |
| R EVALF(String,IER)       | evaluates string                       |
| FLACCS(S,NM,IER)          | gets/sets file access                  |
| FLDATE(NM,DT,IER)         | gets file date and time                |
| FLDIRE(N,D,NM,IER)        | gets names of files in a directory     |
| FLLIST(D,FLIST,N)         | gets list of file names in a directory |
| FLLOAD(FL,B,L,IER)        | load a file into buffer                |
| FLSAVE(FL,B,L,K,IE)       | save buffer to file of type K          |
| FLSIZE(FL,L,IER)          | find the length of a file              |
| FLTIME(S,FL,KT,IER)       | gets/sets the date stamp of a file     |
| FLTYPE(S,FL,K,IER)        | gets/sets file type                    |
| FPSSET(X,U,O,D,I)         | sets floating point exception flags    |

## Application Libraries

## Utilities Library

|                                 |   |
|---------------------------------|---|
| FPSTAT(X,U,O,D,I)               | gets floating point exception flags               |
| GTARGS(STR)                     | gets the command line                             |
| I IGET()                        | single character from keyboard                    |
| I INKEY(NUM)                    | scan keyboard buffer, or OS version number        |
| I JVALK(VAL,KEYW)               | set up keyword for EVALF                          |
| MOUSE(MX,MY,MB)                 | gets pointer position and mouse button            |
| MSGCLS                          | closes the messages file                          |
| I MSGGET(NS,T,<S1..S4>,TXT)     | gets string from messages file with substitutions |
| I MSGGGS                        | as MSGGET but with translations                   |
| MSGOPN(FL,BUF)                  | opens messages file for MSGGET, MSGGS             |
| I MSGSIZ(FL)                    | get buffer size for MSGOPN                        |
| NEXTWD(STR,JS,WORD,LW)          | find next word in a string                        |
| OSBYTE(IFUN,IAR1,IAR2)          | OS_Byte returning no values                       |
| OSBYTE1(IFUN,IAR1,IAR2,IR1)     | OS_Byte returning 1 value                         |
| OSBYTE2(IFUN,IAR1,IAR2,IR1,IR2) | OS_Byte returning 2 values                        |
| L OSCLI(STR)                    | passes string to command line interpreter         |
| OSGETERROR(IER,STR)             | gets cause of error in OSCLI                      |
| OSWORD(IC,IARY)                 | like OS_Byte but uses an array                    |
| PRCLOS                          | close a print job                                 |
| PRINFO(T,V,N,,IX,IY,IXT,IYT,JF) | get information on current printer driver         |
| PRKILL                          | kills the print job immediately                   |
| PROPEN(NAM,O,S,,IB,IE)          | starts a print job                                |
| PRPBEG(NCOPY,IXY)               | starts sending page to printer                    |
| PRPEND                          | ends print page                                   |
| PRSIZE(IPSIZ,IPAREA)            | gets print page dimensions                        |
| RSPACE(SN,NW,[Px])              | allocates dynamic memory for a subroutine         |
| SOUND(IA,IP,LEN)                | makes a musical sound                             |
| TRACEP(N,NAM,LB,LC)             | finds name and locations of routine in trace      |
| VOICE(IV,IST)                   | sets timbre and stereo for SOUND                  |



## Wimp Library

This Library contains a whole system for writing Windows programs in Fortran.

### Introduction

Most Fortran programs have been written with a main program controlling the logical flow. This is not possible in the multi-tasking wimp environment of RISC OS where programs have to share the CPU. Here program flow is controlled by the wimp manager which checks for mouse clicks and key presses, and calls the appropriate task. You can get the manager to call your program to do calculations while all other tasks are dormant.

### Tutorial

The [tutorial](#) and figures at the end of this section show how to write a desktop-compatible program using the Fortran wimp library.

### Library contents

There are two kinds of routine in this library:

- those that you call with names beginning with WP
- routines which are called from the wimp manager via this library and which you can write; these begin with WQ, and all have default entries in the library. The action taken by these default routines is given in the detailed description of each WQxxxx routine in the SrcEdit help file.

The routines are all SUBROUTINES.

### Parameters

Parameters are usually INTEGERS or CHARACTERs. In the documentation, integers follow the standard Fortran default conventions, and character strings all begin with 'S'. The values of the parameters are all input to the routines unless it is specifically stated that the value is 'returned', or an array is to be 'received'. Input parameters come before returned parameters in the argument list.

### Errors

Errors detected by the WPxxxx routines are reported in standard wimp error windows. Usually there is an 'OK' box which allows the program to continue and when the routine will take no action, or return a nonsense value if it has a result. Clicking on 'Cancel' will kill the task immediately.

### Co-ordinate systems

OS units are used throughout, and there are two 'areas' defined for a window:

- the user work area which can be almost any size, and has its co-ordinates measured from the top left corner, hence  $x \geq 0 \geq y$ .
- the visible area which may be displayed on the screen, and can not be bigger than the work area. The co-ordinates are measured from (0,0) at the bottom left of the screen, hence  $x, y \geq 0$ .

The routines in the library use the work area wherever possible, you can use WPXY2S or WPXY2W to transform between the two systems.

**Example Programs**

These are in the Tlibraries directory to show how Wimp programs may be written in Fortran, and also to test the library routines; they have names like Twimpr where n is the number in the list below.

- 1 A very simple program to make a window and display some text
- 2 Shows how to make icons in a window and interact with mouse clicks over them. Also shows how to set up interactive help using !Help
- 3 Puts an icon on the Icon Bar with an associated menu tree including a 'save' box for dragging a file
- 4 Shows how to write a multi-tasking program which does calculations in background
- 5 Tests the rest of the menu routines on the library, and shows how to make a non-multi-tasking text window in a wimp program
- 6 Tests the remaining library routines for manipulating icons
- 7 Changes the screen environment (palette and mode)
- 8 Tests remaining window manipulating routines and dragging areas
- 9 Tests interaction with the task environment, submitting sub-tasks and forced shut-down
- 10 Demonstration of pane windows for tools
- 11 Shows how to use a full screen sized sprite for simplifying graphics
- 12 Calculates and displays a Mandelbrot fractal in a window
- 13 Demonstrates text drawing
- 14 Using the 'Colour Picker'
- 15 Tests the new special icons (WPADNI)
- 16 Tests: WPFTYP, WPGTMS, WPMNWS and WPSTIC

**Routines to call**

These all start with the letters WP. You should not try to write these routines yourself, just call them.

**Initialisation, making windows etc.**

WPADNI WPADSI WPADTI WPBARI WPBART WPCHFT WPCHTSA WPCHSF  
 WPCHTC WPCHTF WPCHCW WPCHWF WPCLST WPINIT WPLDTW WPMKMB  
 WPMKNW WPOPNT WPPANE WPSAVE

**Calling the wimp**

WPLOOP WPMESG WPNULT

**Closing down the wimp**

WPPREQ WPQUIT

**Window operations**

WPCLSW WPCLTX WPCOPY WPDELW WPGTWO WPGTWS WPOPNW WPOPTX  
 WPPLOT WPSETC WPSTTC WPSTWA WPSTWS WPTXT WPTXTC WPTXTF  
 WPTXTW WPUPDT WPXY2S WPXY2W

**Icon Operations**

WPDELI WPDRGI WPGTCP WPGTIF WPGTIL WPGTIS WPRDIN WRRDTX  
 WPSTCP WPSTIC WPSTIF WPSTIS WPWRIN WPWRTX

**Menu Operations**

WPAM2M WPAW2M WPAZ2M WPGTMF WPGTMS WPSHWM WPSTMF

**Error reporting**

WPERR

**General Wimp routines**

WPAINT WPBEGS WPDELS WPDRAW WPDRAW WPEDIT WPENDS WPFTYP  
 WPGETP WPGTPI WPGTSS WPGTTH WPHNDL WPKEYP WPMAKS WPPICC  
 WPSETP WPSTMD WPSTNS WPTASK WPTIME WPUDBX WPVRSN

**Summary of WP routine calling sequences**

|                                |   |
|--------------------------------|---|
| WPADNI(IW,IX,IY,IT.TXT.IC)     | Adds a 'new' type icon to window        |
| WPADSI(IW,IX,IY,SN,IH)         | adds sprite icon to window              |
| WPADTI(IW,IX,IY,JX,JY,S,SV,IH) | adds text icon to window                |
| WPAINT(SFL)                    | sends sprite file to !Paint             |
| WPAM2M(MB2,MB,IND)             | adds menu2 to menu                      |
| WPAW2M(IW,MB,IND)              | adds window to menu                     |
| WPAZ2M(MB,IND)                 | removes sub-menu connection             |
| WPBARI(SN,IP,MB,IH)            | puts sprite icon on icon-bar            |
| WPBART(SN,IP,MB,TXT,IH)        | as above but with text substring        |
| WPBEGS(IW)                     | sends vdu output to window sprite       |
| WPCHFT(IFONT)                  | sets font for text icon                 |
| WPCHSA(IA)                     | changes default sprite area             |
| WPCHSF(IND,X)                  | changes default sprite icon flags       |
| WPCHTC(IC,KOL)                 | changes default text icon colours       |
| WPCHTF(IND,X)                  | changes default text icon flags         |
| WPCHWC(IC,KOL)                 | changes default window colours          |
| WPCHWF(IND,X)                  | changes default window flags            |
| WPCLST                         | closes window template file             |
| WPCLSW(IW)                     | closes window temporarily               |
| WPCLTX(IST)                    | closes the text-only window             |
| WPCOPY(IW,IX,IY,JX,JY)         | forces copy of a rectangle              |
| WPDELI(IW,IH)                  | deletes icon permanently                |
| WPDELS(IW)                     | deletes graphics sprite from window     |
| WPDELW(IW)                     | deletes window permanently              |
| WPDRAW(IBLK)                   | initiates a drag operation              |
| WPDRAW(SFL)                    | sends draw file to !Draw                |
| WPDRGI(IW,IC)                  | initiates dragging an icon              |
| WPEDIT(SFL)                    | ends text file to current editor        |
| WPENDS(IXL,IYB,IXR,IYT)        | ends writing to window sprite           |
| WPERR(IN,STEXT,L)              | makes standard error window             |
| WPFTYP(ITYPE)                  | defines file type for double-click read |
| WPGETP(IPB)                    | gets the colour palette                 |
| WPGTCP(IW,IH,IP)               | gets the caret position                 |
| WPGTIF(IW,IH,IND,X)            | get icon flag                           |
| WPGTIL(IW,IND,LST,NL)          | lists icons with flag                   |
| WPGTIS(IW,IH,IB)               | gets icon state                         |

|   |  |
|---|--|
| WPGTMF(NF,MB,IND,X)                       | gets menu item flag                    |
| WPGTMS(ITEMS,NM)                          | gets state of a menu tree              |
| WPGTPI(IW,IH,IX,IY,IB)                    | gets mouse pointer information         |
| WPGTSS(NCS,NNS,NFP)                       | gets application slot sizes            |
| WPGTTH(TNAME,IHAND)                       | gets task handle of a task             |
| WPGTWO(IW,IX,IY,JX,JY)                    | gets window outline                    |
| WPGTWS(IW,IB)                             | gets window state                      |
| WPHNDL(IHAND)                             | returns the task's handle              |
| WPINIT(SNAM)                              | initialises the wimp                   |
| WPKEYP(KEY)                               | passes on key press information        |
| WPLDTW(SN,IB,SI,IW)                       | loads window from template file        |
| WPLOOP(MASK)                              | starts wimp poll                       |
| WPMAKS (IW,NX,NY,M)                       | creates a sprite in a window           |
| WPMESG(MT,MSG,MD)                         | sends a wimp protocol message          |
| WPMKMB(SL,IB)                             | makes up a menu block                  |
| WPMKNW(IX,IY,JX1,JY1,JX2,JY2,KX,KY,ST,IW) | makes a new window                     |
| WPMNWS(IXMIN,IYMIN)                       | changes minimum size of window         |
| WPNULT(ITIM)                              | sets how often WQNULL is called        |
| WPOPNT(SFL)                               | opens window template file             |
| WPOPNW(IWH)                               | pens window for viewing                |
| WPOPTX(ST)                                | opens a text window                    |
| WPPANE(IWP,IWH,S)                         | attaches a tool pane to a window       |
| WPPICC(IX,IY,S,IR,IG,IB)                  | returns colour from picker             |
| WPPLOT(IW,IX,IY,JX,JY)                    | requests plotting in a window          |
| WPPREQ                                    | continues with wimp shut down          |
| WPQUIT                                    | closes down the wimp                   |
| WPRDIN(IW,IH,INT,IER)                     | reads an integer from a text icon      |
| WPRDTX(IW,IH,S,LEN)                       | reads a string from a text icon        |
| WPSAVE(ST,IS,SN,IW)                       | makes a 'save' window                  |
| WPSETC(IA,KOL)                            | sets colour                            |
| WPSETP(IPB)                               | sets the colour palette                |
| WPSHWM(MB,IX,IY)                          | shows menu                             |
| WPSTCP(IW,IH,IP)                          | sets the caret position                |
| WPSTIC(IW,IC,KOL)                         | changes colour of icon                 |
| WPSTIF(IW,IH,IND,X)                       | sets an icon flag                      |
| WPSTIS(IW,IC,IX,IY,IW,IH)                 | resize/reposition an icon              |
| WPSTMD(MODE)                              | sets the desktop screen mode           |
| WPSTMF(NF,MB,IND,X)                       | sets a menu item flag                  |
| WPSTNS(NNS)                               | sets slot size of next application     |
| WPSTTC(IC)                                | sets colour in text window             |
| WPSTWA(IW,IX,IY)                          | sets work area size                    |
| WPSTWS(IWB)                               | sets window state/size                 |
| WPTASK(SC)                                | starts a new task                      |
| WPTXT(IX,IY,ST)                           | sends text to a window                 |
| WPTIME(RT,CT)                             | returns execution times                |
| WPTXTC(IR,IG,IB,JR,JG,JB)                 | sets colour for text                   |
| WPTXTF(IX,IY,IW,S)                        | sends text to a window in desktop font |
| WPTXTW(S,IWID)                            | gives width of string to print         |
| WPUPDT(IW,IX,IY,JX,JY)                    | requests immediate window update       |
| WPVRSN(IV)                                | returns wimp version number            |

|                        |   |
|------------------------|---|
| WPWRIN(IW,IH,INT,IER)  | writes in integer to a text icon        |
| WPWRTX(IW,IH,S,LEN)    | writes a string to a text icon          |
| WPXY2S(IW,IX,IY,JX,JY) | transform window to screen co-ordinates |
| WPXY2W(IW,IX,IY,JX,JY) | transform screen to window co-ordinates |

**Routines the wimp calls**

These start with the letters WQ. There is a default version of each on the library, but you are expected to write several yourself; e.g. to decide what to do when a mouse key is clicked, and to draw things in windows.

**Summary of calling sequences in the WQ routine calls**

|                            |                                    |
|----------------------------|------------------------------------|
| WQCLIK(IW,IH,IX,IY,IB)     | mouse button click                 |
| WQCLSW(IW)                 | request to close window            |
| WQDRAG(IW,IC,ICOX)         | drag box dropped                   |
| WQHELP(IW,IH,IX,IY,SH)     | request for on-line help           |
| WQKEYP(IW,IH,IX,IY,IC,K)   | key click                          |
| WQLGCT(GAIN,IB)            | losing or gaining caret            |
| WQMENU(MB,ITEMS,NM)        | menu item selected                 |
| WQMESG(MT,MF,MR,MP,MA,MSG) | wimp message received              |
| WQMODC                     | screen mode about to change        |
| WQMODF(JF)                 | warns desktop font has changed     |
| WQMWRN(MB,ITEMS,NM)        | pointer moving over menu arrow     |
| WQNULL                     | null call to do calculations etc.  |
| WQOPNW(IW)                 | request to open window             |
| WQPALC                     | palette changing                   |
| WQPLOT(IW,IXL,IYL,IXH,IYH) | request to draw a window area      |
| WQPREQ(IOK)                | request if shut down OK            |
| WQPTWW(IW,ENTER)           | pointer entering or leaving window |
| WQQUIT                     | request to quit                    |
| WQRFIL(IW,IH,IS,IT,SFL)    | request to read a dragged file     |
| WQSCRL(IWB)                | window scrolling                   |
| WQUPDT(IW,IXL,IYL,IXH,IYH) | request to update a window area    |
| WQWFIL(IW,SFL)             | request to write a file            |



## ***A tutorial on writing a Wimp program***

The Wimp is a useful environment if you want the user to have frequent interaction with the program. If all you want to do is some data analysis it is much more efficient to write a traditional straight Fortran program which takes over the whole computer.

In the following tutorial, frequent reference is made to the supplied example programs which are referred to as 'Example 1' to 'Example 16'. Please use the on-line help of !SrcEdit to find the calling sequences of the routines in the Wimp library or make a listing of it from the file: ...!SrcEdit.help.HelpWimp

The beginning of a Wimp program is quite normal and follows the flowchart shown in [Figure 1](#).

### **User Initialisation**

First do your own initialisation; this may involve setting a lot of variables, reading a file of fixed data or other useful information but not any run-dependent data. These you should obtain later by having a filer icon 'dropped' onto one of your windows.

### **Wimp Initialisation**

Now announce yourself to the Wimp manager by calling WPINIT(NAME) where 'NAME' is an identifying character string for your job which now becomes a Wimp 'task'; it will be listed with this name in Application tasks in the 'task display' window which you get by clicking over the system icon at the right hand end of the icon bar.

### **Create Windows**

Next you must set up some windows or at least put an icon onto the Icon Bar so that the user can work with your task.

There are two ways of creating windows:

- 1 making them up for yourself using WPMKNW. Here you have to specify the size of the window, its position on the screen and its title. Example 1 shows how to set up a very simple window this way and Example 6 shows how you can alter the appearance of a window by changing the parameters of WPMKNW using calls to:
  - WPCHWC to use different colours
  - WPCHWF to remove any of the surrounding borders and their window control icons
  - WPCHSA to allow sprites created in your own area to be displayed
- 2 reading them from a 'Templates' file which you have made previously with a template editor such as !WinEd. This method uses subroutines WPOPNT to open the templates file, WPLDTW to load template windows and WPCNST to close the template file when you have finished reading window definitions from it. Example 4 shows this method.

Each window you create in this way is given a unique 'handle' by the Wimp manager and you will use these when you want to refer to a window for the rest of the program.

### **'Save' Windows**

These are special windows used only for saving files written by your program. They are the familiar little windows which appear when applications like !SrcEdit want you to save its data to a file. In Fortran they are made by calling WPSAVE as shown in Example 3.

**Creating icons in windows**

Very often you will want to make little boxes with words or figures in them in some of your windows which can then be used for making selections or otherwise controlling what your program does. The windows of the IFDE front-end for Fortran are full of examples of these 'Icons'. Again you can either create these at the same time as the window with the template editor or use:

- WPADTI (or WPADNI) to add a 'text' icon to one of your windows as in Example 2
- WPADSI to add a 'sprite' icon as also shown in Example 2 for a Wimp-defined icon (such as the ones in filer windows). Sprites you have designed in your program using the 'SpriteOp' library can be used instead which is shown in Example 6.

Before creating such icons you can change the default text icon colours with WPCHTC (as in Example 6), and change text or sprite icon flags with WPCHTF or WCHSF (as in Example 2). Icon flags control such things as how mouse clicks over the icon are going to be reported to your program, how the text is justified within the icon and whether the icon is transparent or has a border. Each icon in the window is given a unique number or 'handle' by the Wimp manager. These handles start from 0 for the first you create and increment by 1 for each one in the window. It can make the logic of a program much simpler if you order them suitably.

**Creating menus**

Unlike windows, for which the definitions are stored in the Wimp manager, menus are stored in your task's memory - you have to reserve space for them preferably in COMMON so that they can be accessed easily from any routine. Each menu needs six words per entry and an overhead of seven words for the title and other overall definitions. WPMKMB makes up such a menu block from information on the title and entry names (see Example 3)

Menus can be built up into a tree structure by adding a menu to a menu with WPAM2M or a window may be added to a menu item with WPAW2M (Example 3 contains both of these).

You can set the flags on menu items at any time with WPSTMF; these flags control whether the item is ticked, whether it is grey/unselectable and whether it is followed by a dividing dotted line in the list. Example 3 also shows some of these functions.

**Creating an icon on the Icon-Bar**

WPBARI is a special routine for putting your icon on the icon-bar. This is most useful for tasks which spend most of their time doing nothing except waiting for the user to wake them up; Example 3 shows how this is done. Such icons usually have a menu pop-up when you click over them with the middle mouse button. WPBARI expects a reference to one of the menu blocks just created. Routine WPBART also puts text under the icon on the icon-bar.

**Displaying a window**

Although you may now have created lots of windows they will not appear on the screen until you 'open' them. Examples 1 and 2 do not put an icon on the icon-bar but start their dialogue with the user by asking the Wimp manager to display a window. This is done by calling WPOPNW with the appropriate window handle as the argument.



**Starting the Wimp manager**

All the setting-up is done; now your task is ready to interact with the Wimp environment. It does this by calling the Wimp manager through the routine WPLOOP. The Wimp manager now takes over control and calls your task only when the user clicks a mouse button with the pointer over your window, types on the keyboard, drops a filer icon on one of your windows etc. At this point The Wimp manager calls your task through a WQxxxx routine which you have to write. This is shown diagrammatically in [Figure 2](#). This is not as bad as it first appears: WPLOOP is in the middle and the WQxxxx routines it calls surround it - look at one at a time.

**Writing text or graphics to a window**

The Wimp manager may find that the display in one of your windows needs refreshing either because you have just opened it or because some other window has been moved from on top of all or part of it. The manager does not keep track of what should be displayed in a window (apart from icon data), so it clears the area to the window background colour and then calls WQPLOT to ask you fill it in.

You will need to write a WQPLOT for almost any Wimp task unless the window data are all in the form of icons. The co-ordinate system is set up with the origin in the top left corner of your work area so all your co-ordinates will have positive x and negative y. If you want to write text, first use the graphics routine GRMOVE(IX,IY) to set the position to start the text and then use the Fortran PRINT. Alternatively the routine WPTXT(IX,IY,TEXT) prints the character string TEXT at (IX,IY) or WPTXTF(IX,Y,IW,TEXT) prints it in the desktop font. You can use the Graphics library for drawing figures but use WPSETC for changing colours. Example 1 shows how to write text into a window, Example 4 shows a spiral using the graphics library to plot the line segments.

You may also want to replot a window yourself when some information changes. To do this you must not just plot on the screen where you last plotted, the user may have moved the window or stuck something on top of it! You must request the manager to organise it by calling WPPLLOT.

Although the manager may only ask for you to fill in a piece of your window, it is usually easier (but slower) to plot the lot. Only the requested bit will appear because the rest has been masked off.

**Updating the contents of a window**

It may be that you don't want to re-plot the whole window when some changes occur in your data but only to change a little bit. Highlighting a selected item is a good example. Again you must not just go and plot it for the reasons mentioned above. Instead you call WPUPDT to request an update. This in turn calls WQUPDT where you can update your window but unlike WQPLOT the area is not cleared first. This is a much quicker process than replotting the whole window. Your WPPLLOT routine must always be aware of these updates so that if it is called upon to replot the window it does it with all your updates. The WPUPDT-WQUPDT sequence is demonstrated in Example 4 where each new segment of the spiral is plotted as an update.

**Simplified window graphics**

The previous sections showed how to plot text and graphics straight into a window using WQPLOT and WQUPDT. This may be the fastest way to write to a window but it requires that you always remember what you have plotted so that it can be replotted at any time. This may be inconvenient.

Example 11 shows how to make a sprite the full size of a window which covers 800x800 OS units. This sprite is **big** but it is put into the system sprite area so that it does not affect the size of your program. The example shows how to use the graphics library to plot to the sprite. WQPLOT is not used with this method.

### **Mouse clicks over windows**

Almost all Wimp applications will need to respond to clicks from the mouse. The Wimp manager calls WQCLIK when the mouse is clicked over one of your windows telling you which window and which icon it is over which button has been clicked and where. Your program then has to take the appropriate action. Example 2 shows how to stop the program when the mouse is clicked over a specific icon, Example 5 uses a mouse click to open a menu, Example 6 redraws a piece of a window in response to a mouse click. Examples 7 to 10 show more complicated uses of WQCLIK.

### **Mouse clicks over menus**

A mouse click over one of your menus causes the Wimp manager to call WQMENU. The arguments specify the top menu block (of a possible menu tree) and which items in the menu tree are selected by the mouse click. Example 3 shows a typical use of a menu tree attached to an icon on the Icon-Bar; it stops the task when 'Quit' is selected and toggles a tick mark in the menu.

### **Keyboard input**

Normally keyboard input is directed to one window - the one which has 'input focus'. This is usually indicated by having the title bar at the top of the window coloured cream rather than the usual mid-grey. A mouse click over a 'writable' icon in a window will make it have the input focus so that characters may be typed into it. Otherwise you can force one of your windows to have input focus by calling WPSTCP which sets the input caret (the vertical red bar) at any position you like.

### **Input to writable icons**

This is all done by the Wimp manager so that the first you know about it is when the user types <Return>. Your routine WQKEYP is then called with the ASCII code 13 (for <Return>). The user does not have to type <Return> but they have to signal somehow that they have finished typing so that your program can call WPRDTX to read the text into a CHARACTER string (or WPRDIN to read it in as an integer).

### **Direct input to a window**

When your window has input focus and the caret is not in a writable icon, all the key-presses are reported to you through calls to WQKEYP. Here you can do what you like with them. If you were writing an editor you would display them on the screen with a call to WPUPDT (which calls WQUPDT).

### **Hot-keys**

These are keys which the Wimp manager can not handle itself, for example the function and arrow keys. Like all other keyboard input they are handed to the task with input focus first but if it is not interested in them the key-pressed signal should be passed back to the Wimp manager by calling WPKEYP. The manager then passes them to any other window which has the 'hot-keys' flag set. You can set the 'hot-keys' flag when you first create your window with WPCHWF so that you can receive such signals from the Wimp without necessarily having the input focus.

**File handling**

You can read and write files with pre-determined names and disk addresses in a Wimp task in exactly the same way as in normal programs. One of the advantages of running in the Wimp is that file names can be dragged to and from your application.

**Reading files**

A user who wants your task to read a file will drag a filer icon for the appropriate file and drop it on one of your windows or possibly on the Icon-Bar icon. The Wimp manager then calls WQRFIL with the full file name, its size and file type. It also tells you the 'handles' of the window and icon where the file was dropped. In your routine WQRFIL you can OPEN the file and read it in the normal way; Example 9 shows how a file name is received in this way.

**Writing files**

If your task wants to write a file this is best done by creating a 'save' window with WPSAVE (which you would probably do in the initialisation phase of the program) and then calling WPOPNW to display it on the screen. Alternatively you could attach the 'save' window to a menu with WPAW2M. The user will then drag the file icon in this window to a filer directory window (or another wimp task) and drop it. The Wimp manager then calls WQWFIL with the full file name and directory string and the window handle of your 'save' window. Example 3 shows a trivial saving operation with the 'save' window as part of a menu tree.

**Messages**

These are an important part of the Wimp environment. Tasks communicate with one another by sending and receiving messages but the protocol involved is quite complicated.

**Messages handled by the Wimp library**

When your task is reading or writing files initiated by dragging files there are many messages sent to and from the filer task before WQWFIL or WQRFIL is called. Other disguised messages which the wimp library interprets for you are those which call WQLGCT (to tell you you are losing or gaining the caret), WQMODC (for a screen mode change), WQPALC (for a palette change) and WQPREQ (to tell you of an imminent shut-down).

**Open Messages**

Your task may receive other messages through WQMESG. It is best not to write this routine unless you want to have conversations with other tasks. Example 9 shows how two tasks talk to one another using WPMESG to send messages and WQMESG to receive them.

**Background computing**

Most tasks running in the Wimp are driven by mouse clicks and keyboard input (e.g. editors, DTP, painting) and do not use the central processor for long periods. Fortran scientific calculations need not interfere much with other Wimp programs provided that the job can be split up into small chunks. Example 4 shows a trivial calculation of a spiral done in small steps. The Wimp manager has been started in such a way that it calls the user routine WQNULL whenever it has nothing else to do for other tasks. Each call to WQNULL calculates a segment of the spiral and draws it in its window (using WPUPDT - WQUPDT of course!).

**Stopping your task**

This is easy: first save any data (summaries etc.) using 'save' windows and WQWFIL, then call WPQUIT and the program flow continues from where you called WPLOOP after the initialisation.

**Forced stop**

The Wimp manager may decide that your task has to stop. This can either be because the user has requested to shut down the computer or because the 'Quit' has been clicked in the task display window. Provided you don't need any dialogue with the user, the Wimp manager will call WQQUIT when you can do your final tidying up. If you do want to get the user to drag files etc. write a routine WQPREQ to open the 'save' window and write messages using WPUPDT.

**Debugging without !DDT**

It is very unlikely that your program will work perfectly first time and very likely it will end up in a mess so that some method of debugging is needed. Since you are in a Wimp environment you can not use the traditional method of filling the program with PRINT statements: the messages will not appear on the screen. You could write them to a file but that would not allow any interaction. There are two methods of debugging wimp programs with PRINT statements: the first is quite elegant and easy but can not be used while you are plotting in routines WQPLOT or WQUPDT because they would interfere with the writing to the screen; the second is dirty but can be used at any time.

**Text windows**

There is a special kind of window for printing text. This is the same as you get if you run a non-Wimp program from the desktop.

First open the window with WPOPTX, next PRINT your messages, ask for input etc. and then close the text window with WPCLTX. This all must be done in one call from the Wimp manager so you might put the call to WPOPTX at the beginning of your WQCLIK routine and the call to WPCLTX just before the RETURN. Then you could fill WQCLIK and any subordinate routines with PRINT statements.

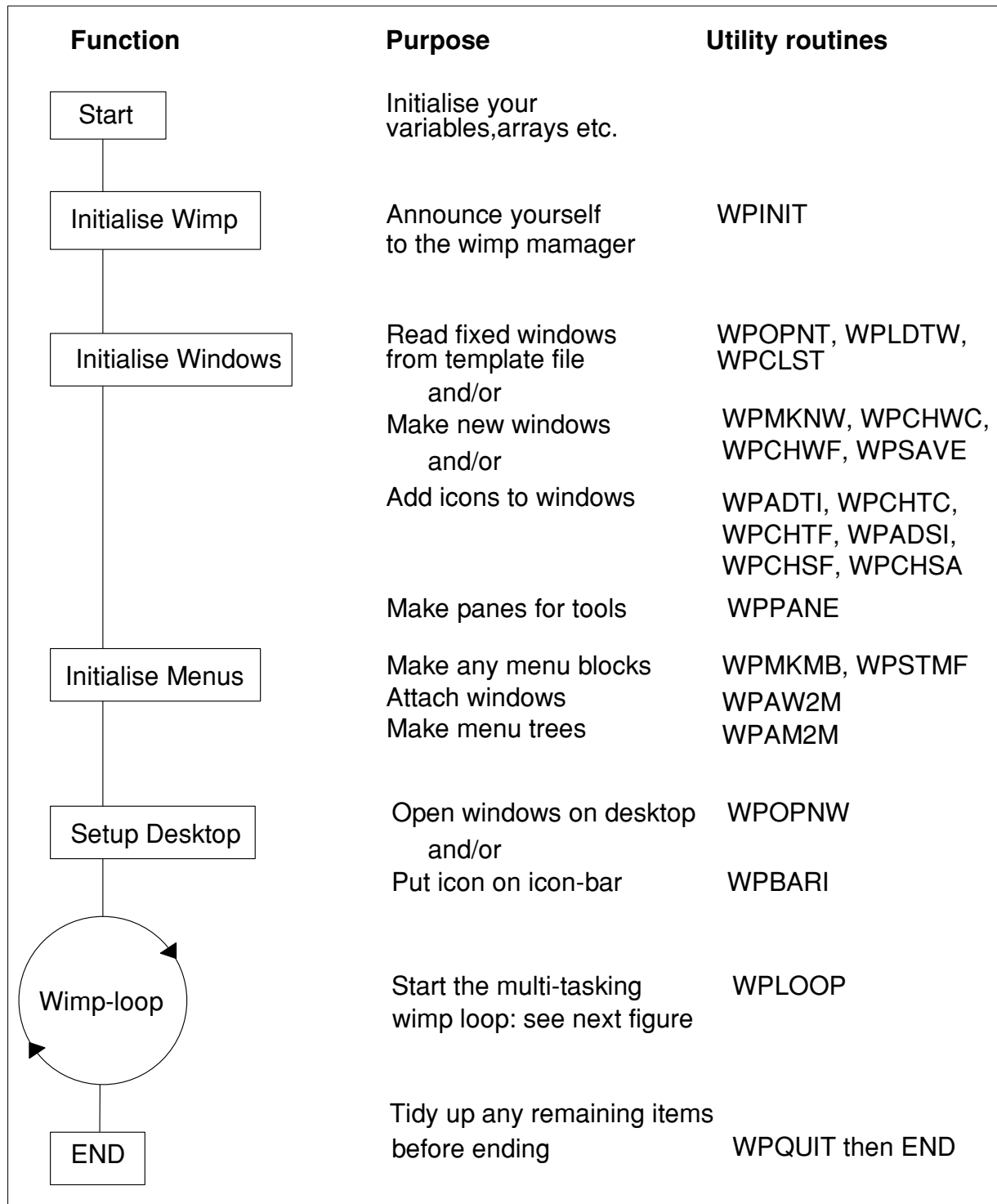
**Immediate writing to the screen**

You can fool the Wimp manager by entering 'text' mode. Normally all the wimp printing and plotting is done as if it were graphics. The following example shows how to go about using text mode:

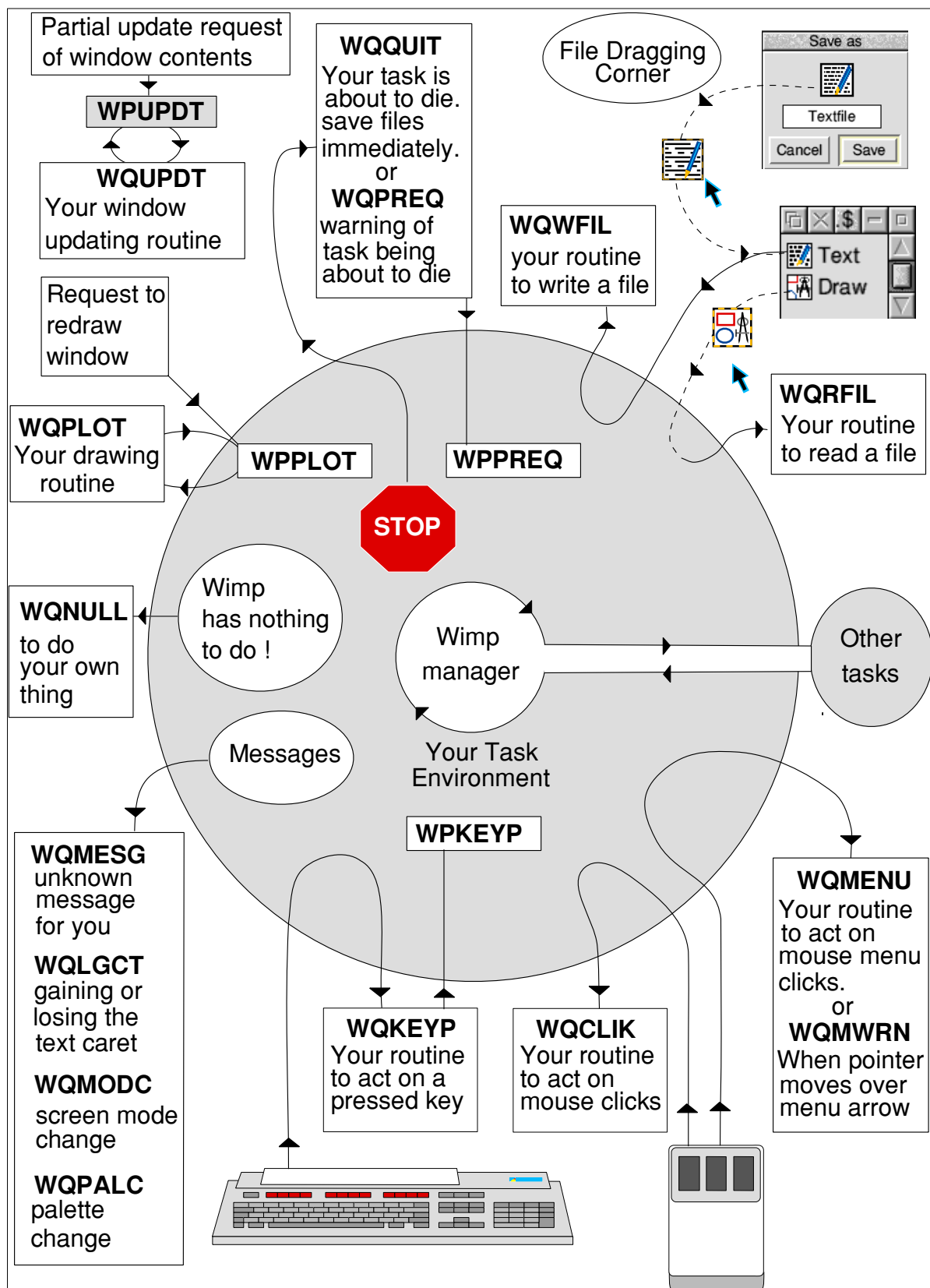
```
CALL GRVDU(4)      !Enter text mode
CALL GRTAB(0,0)    !Tab to the top left of the screen
PRINT *, 'Here is my text etc.'
I = IGET()          !Wait for keyboard input
CALL GRVDU(5)      !Return to graphics mode
```

This works fine even if you are in the process of plotting in a window. The problem is that it writes all over the top of any windows and makes a nasty mess of the beautiful Wimp screen. This is why you are told in the graphics application library not to use GRVDU and GRTAB in a Wimp program.

**Figure 1. Initialisation and termination.  
Flow-chart of a typical Fortran Wimp Program:**



**Figure 2. Interaction with the Wimp manager**  
**Flow-chart of a typical Fortran Wimp Program:**



## **Reference Section**

## **Fortran Desktop Environment**

This is the RISC OS wimp application, !FDE, which allows compiling, linking and running Fortran programs from the desktop.

### **Directory structure**

!FDE assumes that you have working directory set up for your project which has at least an 'f' sub-directory with source files in it. While running, it may set up the following directories:

- 'o' for the object code resulting from compilations
- 's' for assembler listings if you request them
- 'l' for requested compiler source listings
- 'e' for error messages. This will be deleted if there are no errors

Resultant executable image files (type FF8) are stored in the working directory.

### **Icon Bar menu**

Double-click on the !FDE icon in the filer window to mount the icon on the Icon Bar. Click <Menu> to open the top level menu:

| Fortran 77    |  |
|---------------|--|
| info          | ▶ move over arrow to see the version number    |
| help          | click here to see a simple help window         |
| save options  | click here to save the current options         |
| ✓ check dates | click here to toggle date checking (see below) |
| quit          | click here to remove the !FDE application      |

### **Date checking**

The 'check dates' entry in the menu above enables or disables a check on the relative date and time of a Fortran source text file and its corresponding object code file. When it is selected:

- asking to compile a source file which has a corresponding object file of the same or later date/time causes the compilation to be skipped.
- loading an object file into !FDE which is older than the corresponding Fortran source, results in the source being loaded instead.

These features are very useful for a project with many source files. All the source files for the project can then be dragged to !FDE which compiles only those that have changed since a previous compilation.

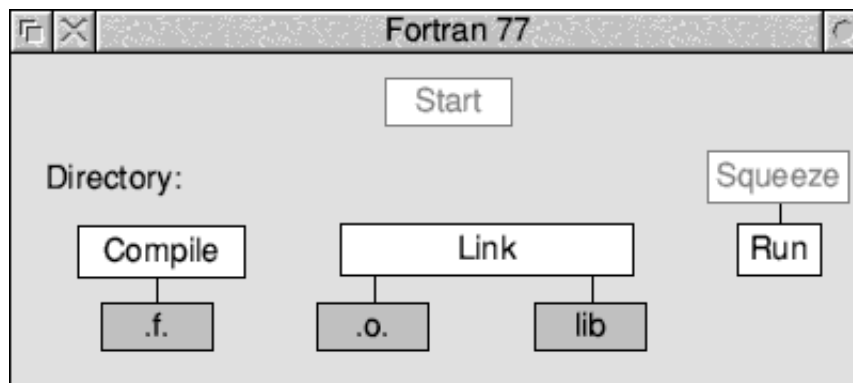
### **Loading files into !FDE**

Drag Fortran source files from your 'f' sub-directory to the icon and they will be shown in the main control window of FDE. Similarly, previously compiled object files may be dragged from the 'o' sub-directory, or an executable image from the working directory. This will display the main control window with the files listed in it. Alternatively, open the window by clicking <Select> over the Icon Bar icon and drag the files into the window.

### **!FDE main control window**

The main window looks like this when it is first opened and no files have been dragged to it:





Fortran source files dragged to it are listed with a cream coloured background under the **.f.** icon on the middle left in the diagram above. Dragged object files from the 'o' directory are listed under the **.o.** icon.

### Working Directory

When any file is dragged to the main window, the working directory name is shown following the word 'Directory:' near the top of the window.

### Selecting the Compiler options

Clicking <Menu> over the 'Compile' icon opens the Compilation options window:

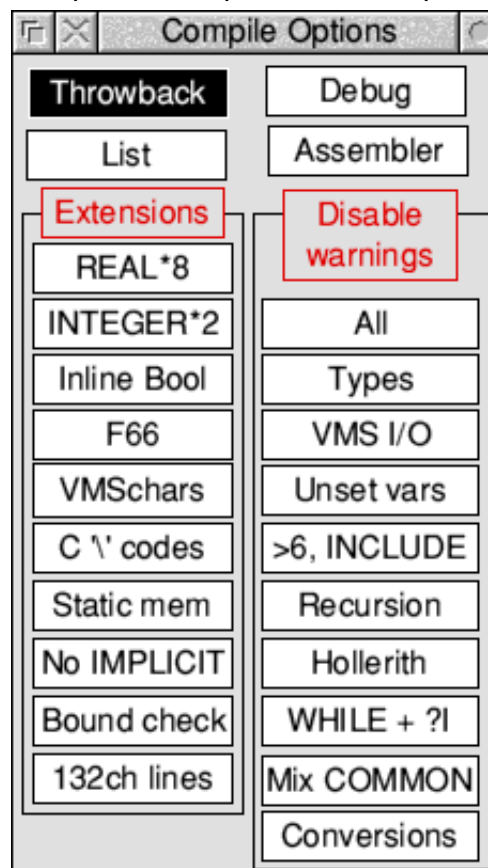
This is a selected option

This is just a heading

Selected options are shown in the window with black background and white lettering .

Use <Select> to select an option and <Adjust> to deselect it.

The options *do not toggle*



An explanation of these options is given in the following table (except for [Throwback](#) which is explained later):

| Option <sup>1</sup>   | Menu icon  | Action   |
|---|--|--|
| -g<br>-list<br>-s   | Throwback<br>Debug<br>List<br>Assembler  | interactive compilation error correction<br>insert debug code for !DDT<br>write a source listing in l. <i>filename</i><br>make an assembler listing in s. <i>filename</i> instead of object code in o. <i>filename</i>   |
| -zpx22#<br>-zpx21#<br>-zpx6#<br>-zpx11#<br>-zpx13#<br><br>-zpx27#<br>-zpx20#<br>-zpx23#<br>-zpx24#<br>-zpx26# | REAL*8<br>INTEGER*2<br>Inline Bool<br>F66<br>VMSchars<br><br>C '\ ' codes<br>Static mem<br>No IMPLICIT<br>Bound check<br>132ch lines | make all REAL variables REAL*8<br>make all INTEGER variables INTEGER*2<br>make ISHFT, IOR etc. into in-line code<br>allow some Fortran66 constructs<br>allow VMS-type in-line comments using '!', also '_' and '\$' in the allowed character set<br>treat '\ ' as in UNIX/C escape sequences<br>make all variables static (like SAVE all)<br>invent IMPLICIT NONE for all routines<br>insert code to check subscripts<br>allow lines up to 132 characters (instead of the standard 72) |
| -w<br>-zpx0#<br>-zpx1#<br>not -fa<br>-zpx5#<br><br>-zpx7#<br>-zpx9#<br>-zpx10#<br>-zpx12#<br><br>-zpx14#      | All<br>Types<br>VMS I/O<br>Unset vars<br>>6, INCLUDE<br><br>Recursion<br>Hollerith<br>WHILE + ?I<br>Mix Common<br><br>Conversions    | suppress all warnings<br>no warnings for non-standard variable types<br>no warnings for VMS special I/O<br>no warnings for variables used before setting<br>no warnings for long names<br>allow INCLUDE<br>no warnings for routine calling itself<br>no warnings for Hollerith constants<br>no warnings for block DO and hex. constants<br>no warning for mixed variable types in COMMON<br>no warnings for non-standard conversions   |

A complete list of [-zpx](#) options can be found later in this reference chapter

### Compiling with !FDE

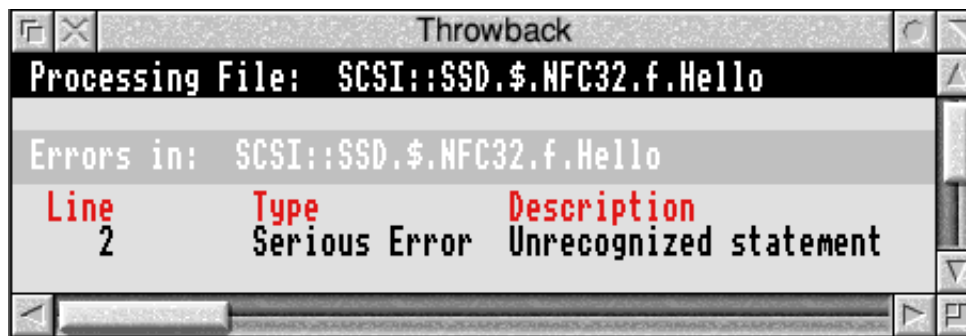
Click <Select> over the 'Compile' icon above the list of source files to compile and then Click 'Start'

### !FDE Throwback

If the 'Throwback' option is not selected and there are errors in the compilation, a standard Wimp error box appears. Clicking on 'Cancel' causes an editor window to open showing the file in the 'e' directory containing the error messages.

With the 'Throwback' option selected and provided that !SrcEdit has been loaded at some time in the session, compilation errors are reported in a window like this:

This is an easy way to find lines with errors. Double-click over an error line in this window and !SrcEdit opens a window on the faulty file with the problem line displayed and highlighted ready for correction.



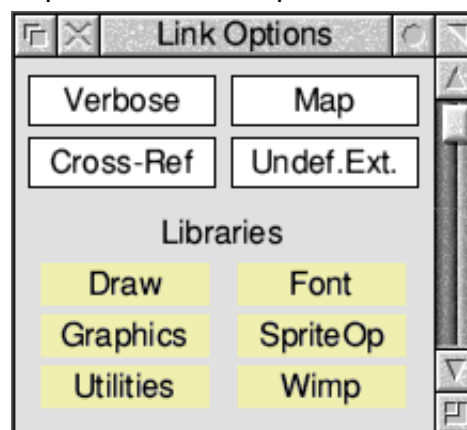
### Selecting the Link options

Clicking <Menu> over the 'Link' icon opens the 'Link Options' window:

When options are selected they are shown with black background and light lettering in the window.

Use <Select> to select an option and <Adjust> to deselect it.

The options *do not toggle*.



All the options shown here are:

| Option | Menu icon  | Action  |
|--------|------------|---|
| -v     | Verbose    | Write informational messages to e. <i>filename</i>    |
| -map   | Map        | Write a link map to e. <i>filename</i>                |
| -x     | Cross-Ref  | Write a cross-reference listing to e. <i>filename</i> |
| -u     | Undef.Ext. | Link even if there are undefined symbols              |

### Libraries

The available libraries (other than the Fortran system library) are listed in icons with cream background underneath the word 'Libraries'. Selecting one also lists it under the **lib** icon in the main window. Libraries must be linked in the right order. If there is a reference from one library to a routine in another, the subordinate one must come lower in the list. The Fortran system library (Fortlib) is always linked last (and does not appear in the list) because it does not refer to any other libraries. The six application libraries supplied with !FDE are independent and can be linked in any order. For example, suppose you write a set of routines for plotting graphs which use the graphics library for their output, it would be useful to make them into a library so that many of your applications could use them. You would make the library with !LibFile<sup>2</sup>, storing the result in the directory with all the other libraries. Then, next time you load !FDE on to the Icon Bar, your high-level graphics library will appear in the Link Options window. Your new library must be selected before 'Graphics' so that it appears above it in the list of libraries under **lib** in the main window. The lists can be reordered (see below).

<sup>1</sup> These options are described in more detail in the [Compiler options](#) section below

**Linking with !FDE**

Click <Select> over the 'Link' icon above the list of object and library files to be linked and then Click 'Start'. This will create the executable image with the name under the 'Squeeze' and 'Run' icons. Its name can be changed using the Run Options (below).

**Squeeze in !FDE**

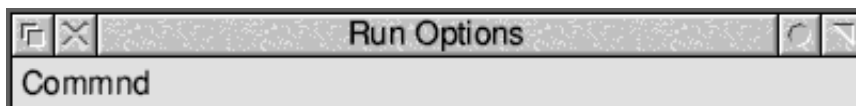
This program compresses the image file, often to about half its full size. Click on its icon in the main window when there is an image file name listed below it and next time you click 'Start' the image file will be squeezed.

**Running a Fortran program from !FDE**

Either drag an image file from your working directory to the Main window or use 'Link' to make the image file listed under the 'Squeeze' and 'Run' icons. Select 'Run', then click 'Start'. The only advantage of running an image this way rather than double-clicking over it in the filer window is that !FDE tries to calculate and set the memory it is going to need before loading it.

**Run options**

Select and click <Menu> over the selected 'Run' icon to open the Run options window:



Here any commands may be entered to send to the program on the command line. In this example 'Commnd' is the name of the executable image. The name of the executable image file can also be changed here by deleting it and typing a new one. To do this either the file must exist or a link step must be requested.

**Removing and reordering FDE files**

File names can be removed from the lists in the main window by clicking <Adjust> over them. With more than one name in a list, clicking with <Select> over one moves it to the top of the list so the order in which files are compiled or linked may be changed.

**Directory aliases**

Two aliases are set up by !FDE:

- |             |  |
|-------------|--|
| F77cl\$Dir  | is the directory where the compiler, linker and squeeze executable images are stored. This will usually be the 'lib' directory within the !FDE application.  |
| F77lib\$Dir | is the directory containing all the Fortran libraries; at least Fortlib must be stored there together with any of Draw, Font, Graphics, SpriteOp, Utilities and Wimp. User libraries should be stored here too. Apart from Fortlib, their names will be displayed in the Link Options window. This will usually be the 'libf77' directory within the !FDE application. |

***Calling the compiler from the command line***

To compile a program with the computer in the command line mode or from a task window, first set your working directory to be the current directory then type:

```
f77 [options] file1 file2 ... fileN
```

where file1 etc. are source files to be compiled. There is no need to prefix them with 'f.' if they are in your 'f' directory.

**File and directory structure**

The compiler will expect to find the following in your working directory:

| Item  | Contents   |
|---|--|
| f directory<br>o directory<br>l directory<br>f77lib.o.fortlib | containing source files<br>to receive the compiled object<br>for optional listings<br>the Fortran system library unless you use<br>the -c option (below) |

**Compiler options**

These are displayed with the command:

```
*f77 -help
```

| Option                  | Action  |
|-------------------------|---|
| -help                   | outputs this list of options  |
| -list                   | Generate a compilation listing in the file <i>l.file1</i>   |
| -c                      | Do not link the files being compiled. Without this option the compiler will try to link your object files using the libraries listed in the '-L' option and the system library fortlib; you can also include object files in the list and they will be linked.<br>!FDE always uses '-c' because it uses a separate link step. Do not confuse this with the -C option below. |
| -throwback              | Support error processing by Desktop Tools & other compliant tools   |
| -desktop <dir>          | Set the work directory for the compiler as <dir>  |
| -E                      | Preprocess the F77 source code only - do not compile or link it   |
| -C                      | Prevent the preprocessor from removing comments (use with -E)   |
| fa                      | Give warnings for variables which may be used before being set.   |
| -g                      | Generate code that may be used with the debugger  |
| -l <libs>               | Specify a comma-joined list of libraries to be linked if the '-c' option is not specified. !FDE never uses this.  |
| -o [file]               | Send the output to [file] (if [file] is omitted a default 'o.file1' is used). Do not confuse this with the -c option above..  |
| -via <file>             | Read in extra command line arguments from <file>  |
| -errors <file>          | Write error output to <file>  |
| -s                      | Write out assembly code instead of object code  |
| -strict<br>or<br>-fussy | Require upper case for all Fortran statements other than those within parentheses or in Hollerith. This makes all references to subprograms into upper case so this option must be applied to all procedures if it is used on any. The application libraries are not accessible using this option. 'strict' works by turning off the zpx options 2 and 3.                   |
| -f66                    | instructs the compiler to enforce the FORTRAN 66  |

|          |   |
|----------|---|
| -i2      | standards whenever there is a conflict with FORTRAN 77 make the default integer size 16 bits (INTEGER*2) rather than the usual 32 bits (INTEGER*4)  |
| -r8      | make the default real size 64 bits (REAL*8) rather than the usual 32 bits (REAL*4)  |
| -extend  | allows FORTRAN lines to extend to 132 characters rather than being truncated at 72  |
| -onetrip | Change the semantics of DO loops so they are obeyed at least once, as was expected in FORTRAN 66. Note that the -f66 flag affects the syntax of the parsed language. This flag modifies the meaning of a DO loop. |
| -w       | Turn off all warning messages   |
| -U       | Do not convert upper case letters to lower case. This means that identifiers become case sensitive  |
| -Otime   | Optimize for small image size   |
| -Ospace  | Optimize for faster execution   |
|          | Experience shows that neither of these optimization options is beneficial!  |
| -zpx<n># | Select option <n> from the list in the table below.<br>The '-zpx<n>#' options can be combined. For example:<br>-zpx1# -zpx5# is equivalent to: -zpx34 (where the 34 is the sum of $2^1$ and $2^5$ ).              |

There are also more technical options used for different ARM processors; the definitions of these can be found in the DDE package documentation from [ROOL](#).

**-zpx type options**

| Option | Action  |
|--------|---|
| 0      | turn off warnings for non-standard variable types e.g. BYTE, DOUBLECOMPLEX etc.   |
| 1      | turn off warnings for VMS special I/O (ACCEPT,TYPE, DECODE and ENCODE)  |
| *2     | Make lower case Fortran code into upper case  |
| *3     | Swap case of Fortran code (hence making it all lower case and compatible with the libraries)  |
| 4      | <i>unused</i>   |
| 5      | turn off warnings for using double rather than single quotes and for >6 character names. Also allow INCLUDE                             |
| 6      | make logical/boolean functions into in-line code  |
| 7      | turn off warnings about direct recursive calls  |
| 8      | <i>unused</i>   |
| 9      | turn off warnings when variables are initialised with Hollerith   |
| 10     | allow block DO, WHILE, DO WHILE and hexadecimal constants using ?I  |
| 11     | allow some Fortran66 constructs (SUBROUTINE MAIN for PROGRAM etc.)  |
| 12     | turn off warnings about mixing CHARACTER with other variable types in COMMON (does not turn off message about padding if it is needed). |
| 13     | allows the VMS extensions: '!' in-line comments, '_' and '\$' in the allowed character set  |
| 14     | turns off warnings for non-standard assignments e.g. LOGICAL=string, INTEGER=LOGICAL etc.   |
| 20     | make all variables static (like SAVE everything but this degrades the optimising)   |
| 21     | make all integers into INTEGER*2 (or use option -i2)  |
| 22     | make all reals into DOUBLE PRECISION and COMPLEX into COMPLEX*16 (or use option -r8)  |
| 23     | apply IMPLICIT NONE (A-Z) to all routines.  |
| 24     | check subscripts are within bounds by inserting code to check each instruction which uses indexing during execution                     |
| 25     | <i>unused</i>   |
| 26     | allow lines of up to 132 significant characters instead of the standard 72 (or use option -extend)                                      |
| 27     | treat '\ ' as in UNIX escape sequence (e.g. \n = newline)   |

Note the options marked \* are set by default and may only be unset by using the main option '-strict'.

### Calling the linker from the command line

A program compiled from the command line has to be linked with the necessary libraries before it can be run. The command line for this is:

```
link [options] object_file1 object_file2... library_file1
      library_file2...
```

As many object files and library files as required may be linked but they may need full path names. For example:

```
link -o Myprog o.Myprog %.libf77.fortlib
```

will link the compiled object file 'o.Myprog' with the Fortran system library to make the executable file 'Myprog' (assuming that the current directory is the project working directory and the Fortran libraries are stored in a directory 'libf77' in the system library).

#### Link Options

Useful options for use with linker are:

| Option                | Action                                     |
|-----------------------|--|
| -o <i>image_file</i>  | Put output in 'image_file'                 |
| -s <i>symbol_file</i> | List symbols to 'symbol_file'              |
| -v                    | Print informational messages while linking |
| -map                  | Print link map                             |
| -x                    | Print area cross reference                 |
| -c                    | Ignore case when symbol matching           |
| -d                    | Include debugging information              |
| -via <i>via_file</i>  | Take file names from via_file              |
| -u                    | Link even if there are undefined symbols   |

A complete list of options may be obtained by typing:

```
*link -help
```

### Running jobs from the command line

Compiling and linking a program will produce an executable image defined by the '-o' option in the link step. (If FDE was used to make the image file it will be in the project working directory and have the name shown in the main FDE window).

To run this program type:

```
*Filename [options]
```

Any options can be read in to the program using SUBROUTINE GTARGS from the Utilities library.



## Fortran 77 Extensions

Since Fortran 77 became standardized (ANSI X3.9-1978) many compilers have extended this standard. Fortran for the RISC OS computers also offers enhancements, many of which are also available on other platforms.

### The Fortran Character set

In addition to the standard Fortran character set, lower case letters are allowed but no distinction is made between these and the upper case letters. For example, the symbolic names: MYWORD and MyWord will refer to the same variable. Within quotes in CHARACTER variables or in FORMAT statements the case is preserved as it is also in Hollerith constants.

The ASCII characters '\_' and '\$' are also allowed if the zpx13# option is selected ('VMSchars' in FDE).

### Code-line comments

Comments may be placed after Fortran code on the same line by prefixing them with '!' and using the option -zpx13# thus:

```
CALL GRCIRC(IX,IY,IR,.TRUE.) !draw a solid circle
```

### Continuation lines

Up to 99 continuation lines (character in column 6) are permitted in contrast to the standard maximum of 19.

### Symbolic names

These are not restricted to the standard 6 characters but may extend up to 255 characters, each of which is significant. Select option -zpx5# to suppress warnings for these ('>6, INCLUDE' in FDE).

### Hexadecimal Constants

Constants may be entered in hexadecimal notation by prefixing them with '?I'. This can only be used directly for loading integers, however by using EQUIVALENCE they may be loaded into any variable type since even character variables can be equivalenced to integers. For example:

```
EQUIVALENCE(R,K)
K = ?I7F800000
```

will load +infinity into the REAL variable R. The ?I notation can also be used to load hexadecimal constants in DATA statements.

### Hollerith constants

Hollerith data can be used not only in DATA and FORMAT statements but also in arithmetic and logical statements. For example:

```
DOUBLE PRECISION WORD
C      this statement fills the 8 bytes of WORD
C      with the ASCII data for 'Holler' followed
C      by two blanks
WORD = 6HHoller
```

The character count before the 'H' must correspond to the number of following characters and not to the length of the word being filled so there must be no significant trailing blanks. The bytes are filled in the same order as CHARACTER data so that the above code has the same effect as:

```
CHARACTER*8 ASCII
EQUIVALENCE(ASCII,WORD)
ASCII = 'Holler'
```

Set option -zpx9# ('Hollerith' in FDE) to avoid warnings when using Hollerith data.

**Other non-standard assignments**

Other possible allowed assignment statements are:

```

LOGICAL LOGI
INTEGER INTG
REAL FLT4
LOGI = 'String'
INTG = 'String'
INTG = LOGI
LOGI = INTG
FLT4 = LOGI
FLT4 = 'String'

```

In such statements the INTEGER and LOGICAL types are equivalent with no conversion between them. LOGICAL to REAL conversions are then just like INTEGER to REAL but 'String' to REAL has no floating conversion.

**IMPLICIT NONE**

This extension is implemented with the option '-zpx23#' ('No IMPLICIT' in FDE) and causes an 'IMPLICIT NONE' in all compiled routines; it forces an explicit type statement for all used variables. This option can be useful for detecting mistyped symbolic names in a program since all expected symbolic names have to be defined before they can be used.

**Variable types**

Additional variable types:

- BYTE are 8-bit integers, defined to have values between -128 and 127 when converting to other INTEGER types
- INTEGER\*2 are 16-bit signed integers. Although these save a lot of space for large arrays of small integers, the RISC architecture only allows for 8- or 32-bit words which makes the object code *much* more complicated. These should not be used for frequently performed integer arithmetic.
- COMPLEX\*16 (or DOUBLE COMPLEX) are complex numbers like type COMPLEX but made from two DOUBLE PRECISION rather than REAL words.

Synonyms:

- REAL\*4 for REAL
- INTEGER\*4 for INTEGER
- LOGICAL\*4 for LOGICAL
- COMPLEX\*8 for COMPLEX
- REAL\*8 for DOUBLE PRECISION

are accepted by the compiler, (use option -zpx0# ('Types' in FDE) to stop warnings).

**Complex\*16 arithmetic**

The rules governing COMPLEX\*16 are exactly the same as those for type COMPLEX where DOUBLE PRECISION numbers are used instead of REAL. For example:

```

COMPLEX*16 Z2,W2
C      take double precision complex square root
W2 = SQRT(Z2)
C      take its complex conjugate
W2 = CONJG(W2)

```

The intrinsic functions with COMPLEX\*16 arguments are shown in the following table:

| Generic name | Specific name | Action             | Result     |
|--------------|---------------|--------------------|------------|
| ABS          | CDABS         | Magnitude          | REAL*8     |
| CMPLX        | -             | Truncate precision | COMPLEX*8  |
| CONJG        | DCONJG        | Complex conjugate  | COMPLEX*16 |
| COS          | CDCOS         | Cosine             | COMPLEX*16 |
| DBLE         | DREAL         | Real part          | REAL*8     |
| -            | DIMAG         | Imaginary part     | REAL*8     |
| EXP          | CDEXP         | Exponential        | COMPLEX*16 |
| INT          | -             | Integer real part  | INTEGER    |
| LOG          | CDLOG         | Natural logarithm  | COMPLEX*16 |
| REAL         | -             | Real part          | REAL*4     |
| SIN          | CDSIN         | Sine               | COMPLEX*16 |
| SQRT         | CDSQRT        | Square root        | COMPLEX*16 |

The function DCMPLX results in a COMPLEX\*16 value and takes one or two arguments. If there is one INTEGER or REAL argument, the imaginary part is taken as zero; if there are two INTEGER or REAL arguments they form the real and imaginary parts and if there is one COMPLEX argument it has the precision of both parts increased to REAL\*8.

### Block DO statements

DO loops may be terminated with an unlabelled ENDDO statement, for example:

```

      DO I=1,10
        DO J=1,3
          . . . . .
          XYZ(I,J) = A(J)*COS(T(I))
        END DO
      END DO

```

### WHILE loops

There are three ways to write loops which execute only while some logical expression is true. They are:

```

      WHILE (LOGICAL) DO
C      calculations done while LOGICAL is .TRUE.
      . . . . .
      END WHILE
C      control comes here when LOGICAL is .FALSE.

```

or

```

      DO WHILE (LOGICAL)
C      calculations done while LOGICAL is .TRUE.
      . . . . .
      END DO
C      control comes here when LOGICAL is .FALSE.

```

or

```

      DO 10 WHILE (LOGICAL)
C      calculations done while LOGICAL is .TRUE.
      . . . . .
10 CONTINUE
C      control comes here when LOGICAL is .FALSE.

```

**Equivalence**

CHARACTER type variables may be equivalenced to arithmetic variables but be careful that this does not try to align variables off the natural 4-byte boundaries. Hence:

```
CHARACTER*10 LIST
EQUIVALENCE (K1,LIST(1:1)), (K2,LIST(7:7))
```

will cause the serious error: *Incorrect alignment of 'k2' in equivalence* whereas:

```
CHARACTER*10 LIST
EQUIVALENCE (K1,LIST(1:1)), (K2,LIST(5:5))
```

compiles without complaint because the integers are equivalenced four bytes apart.

When an array of type BYTE words are equivalenced to INTEGER the first BYTE is the least significant byte of the first INTEGER, the fourth is the most significant byte of the first INTEGER, the fifth is the least significant byte of the second INTEGER and so on. INTEGER\*2 words follow the same pattern using 16 instead of 8 bits.

**Mixed COMMON**

CHARACTER variables may be mixed with arithmetic variables in COMMON. It is important not to misalign the arithmetic variables off their 4-byte boundaries. Thus:

```
CHARACTER STR1*6, STR2*8
COMMON /C1/STR1,I1,R1
COMMON /C2/STR2,I2,R2
END
```

gives:

*Warning: Common block 'c1\_\_' requires non-portable padding*

meaning that two padding bytes have been added between the STR1 string and I1. Set option -zpx12# ('Mix COMMON' in FDE) to turn off standard warnings for mixing variables in common though it will not turn off the warnings about padding which are potentially dangerous.

**INCLUDE statement**

This statement allows you to insert code from another file which can be very useful if you have a lot of routines using the same declaration statements. It is then easy to change COMMON (for instance) when you have these statements in a separate file and it will change in all routines. The format of the statement is:

```
INCLUDE 'filename'
```

Which inserts the code from 'filename' into your source at that point.

If you do not enter a full pathname in the 'filename', the compiler will assume that it is in your working directory. Provided your 'f' directory is not too full it is convenient to keep your 'INCLUDE' files with the rest of the Fortran source, so the statements will look like:

```
INCLUDE 'f.filename'
```

Set option -zpx5# ('>6, INCLUDE' in FDE) to allow INCLUDE.

Two reservations: the /LIST qualifier and nested INCLUDEs, allowed in VAX Fortran are not permitted here. (Use the [pragma](#) #include to allow nesting).

**Bit manipulation functions**

There are eight bit manipulation functions which conform to ANSI/ISO Fortran extensions and are defined in the MIL-STD-1753 standard. In these the bits are numbered starting with the least significant (or rightmost) bit as bit 0; I and J are integers.

| Function   | Definition   |
|------------|--|
| BTEST(I,J) | Tests bit J of I: LOGICAL function returns .TRUE. if it is set to 1 or FALSE if it is 0    |
| IAND(I,J)  | Returns logical AND of I and J   |
| IBCLR(I,J) | Returns I with bit J set to 0  |
| IBSET(I,J) | Returns I with bit J set to 1  |
| IEOR(I,J)  | Returns the exclusive OR of I and J  |
| IOR(I,J)   | Returns the logical OR of I and J  |
| ISHFT(I,J) | Returns I shifted left J bits with zero fill<br>the shift is to the right if J is negative |
| NOT(I)     | Returns the logical NOT of I   |

All of these exist as entry points in the 'Fortlib' system library but they can be compiled into your program by selecting the option -zpx6# ('Inline Bool' in FDE).

Also in the system library is the additional MIL-STD-1753 standard function IBITS(I,J,K) which extracts the K bits from word I starting at bit J, returning it in the least significant bits of the word.

### Other library extensions

A few other functions are available in the system library:

| Function       | Definition   |
|----------------|--|
| CDATE(String)  | Returns in String the date and time in format ddd,nn mmm yyyy.hh:mm:ss   |
| CPU(TIME)      | Returns the time in seconds since 'TIME'   |
| ISHFTC(I,J,N)  | Returns I with least significant N bits circularly left-shifted by J (other 32-N bits unaffected)  |
| LNBLNK(String) | Returns the position of the last non-blank character in a string.  |
| LOC(WORD)      | Returns the byte-address of WORD   |
| LOCC(String)   | Returns the address of character String  |
| RND01()        | Returns a pseudo-random number in the range 0.0 (inclusive) to 1.0 (exclusive)   |
| SETRND(I)      | Initiates the repeatable random number sequence I (for I>0) from RND01 or an unrepeatable sequence if I=0. If SETRND is not called, RND01 produces sequence 1. |

### Recursion

It is possible to write recursive code provided that all the variables which are expected to be different at each recursive step are in dynamic memory. This is usually the case unless they are stored in COMMON, are initialised with DATA or are arguments of a SAVE statement.

For example, to draw a set of coloured filled circles:

```

PROGRAM CIRCLES
C          set up initial values
C          centre (x,y)
      IX = 512

```

```

      IY = 512
C      radius
      IR = 500
C      colour
      IC = 7
C      clear screen
      CALL GRCLRG
C      draw circle
      CALL SUB1(IX, IY, IR, IC)
      STOP
      END

C
      SUBROUTINE SUB1(IX, IY, IR, IC)
C      colour IC
      CALL GRGCOL(0, IC)
C      draw circle centre (IX,IY), radius IR
      CALL GRCIRC(IX, IY, IR, .TRUE.)
      IF(IR.LT.100) RETURN
C      recursively draw a smaller circle
      CALL SUB1(IX-4, IY-4, IR-8, MOD(IC+1,16))
      RETURN
      END

```

Set option -zpx7# ('Recursion' in FDE) to turn off compiler warnings for recursive code.

### Index bounds checking

Indices to arrays can be checked at compile and execution time.

The compiler always checks that there are no explicit requests for members of an array outside its dimensions . For example:

```

      DIMENSION A(3)
      CALL SUB(A(4))
      END

```

gives:

*line 2: Warning: out-of-bound offset 4 in address*

This warning can only be turned off by selecting the '-w' option.

In addition, when option -zpx24# ('Bound check' in FDE) is turned on, the compiler inserts code to check that bounds are not exceeded during execution, so:

```

      DIMENSION A(3)
      B = 0.
      DO 20 I=1,6
        A(I) = I*I
        B = B + A(I)
20  CONTINUE
      PRINT *,B
      END

```

on execution gives:

*\*\*\* Range check failure, index = 4 \*\*\**

Inserting this extra code makes the program larger and slower so use this option only for debugging and not for a production program.

### Input / Output Extensions

There are many extensions to the standard Fortran 77 input/output statements included in the FDE library

- 1 Extended format descriptors
  - Zw.d hex (already in standard system)
  - Ow.d octal
  - Bw.d binary
  - Q inputs the size in bytes of the formatted buffer remaining as a 4-byte integer (ignored on output).
  - \$ don't do <newline> after writing to screen
- 2 Stream numbers must be between 1 and 99. Usually stream 5 is reserved for keyboard input (e.g. READ \*,K) and stream 6 for screen output (e.g. PRINT \*,J).
- 3 Format controlled input
  - Numeric/logical input terminates on finding a comma in the data stream. but...
  - a) don't use tab descriptors
  - b) does not work for A descriptor.
- 4 List-directed CHARACTER input: this is normally like 'abcde...', but can be "abcde...", or even abcde..., i.e. without any quotes but in this form the string can not contain a space, a comma, a '/' or have a count thus:
  - 3\*'empty' or 3\*"empty" will transfer three strings 'empty', whereas
  - 3\*empty will transfer one string '3\*empty'.

These forms allow normal comma-separated-variable records to be read without complicated decoding though you must know a priori how many fields there are in a record or terminate all records with "/".
- 5 List-directed complex input: the real and imaginary parts must still be given within parentheses but either or both may be omitted. Thus "(1.0)" or "(1.0,)" will enter a real part of 1.0, leaving the imaginary part unchanged; "(,1.0)" will enter an imaginary part of 1.0 while leaving the real part unchanged.
- 6 List-directed I/O: this is permitted to/from an internal file (CHARACTER variable or first element only of an array since there is no concept of end-of-record).
  - The output data for each variable will start with a blank except for the first variable sent to an internal record.
  - Input is terminated on finding a '/' character or when the list is satisfied; otherwise more records will be read. Similarly output overflow will be written to extra records.
- 7 Formatted file record length: external files, whether sequential or direct-access, have a maximum record length of 512 bytes. The length of records of internal files is limited only by the size of the CHARACTER variable. Unformatted records, whether sequential or direct access, can be any size up to 2 Gbytes.
- 8 Direct-access and unformatted files are incompatible with those created by the standard ([ROOL](#)) library.

9 Indexed Input: if you want to read records in a sequential file in a non-sequential order you can set up an index to the records and then use the index to access them immediately. This works for both formatted and unformatted records as described here:

First you must set up the INTEGER\*4 index array for the file. Do this by reading the file sequentially and before each READ statement:

```
CALL FLGETP(NUNIT,INDX)
```

where NUNIT is the Fortran unit number and INDX is the returned address of the record about to be read. Store this for any records you may want to read non-sequentially later in your index array.

To read a record with index INDX

```
CALL FLSETP(NUNIT,INDX)
```

to position the file at byte INDX; then read the record in the usual way.

10 End-of-file: the ENDFILE statement writes a special record in the physical file which is interpreted on reading and can be detected by IOSTAT=-2 (rather than IOSTAT=-1 for the physical end of the file). You can read beyond this type of end-of-file marker; it counts as a record if you then BACKSPACE over it.

```
CALL FLSKIP(NUNIT,IEOF)
```

skips the rest of the file. It returns IEOF:

- > 0 if the file is not open or is illegal in any way
- =-1 for physical end-of-file detected
- =-2 for logical end-of-file (as described above) and positions the file for reading after the marker.

11 OPEN qualifiers

- FORM='PRINTER' to interpret Fortran carriage control characters (implies FORM='FORMATTED'). The first character of each record is interpreted thus:

| Character | Function  |
|-----------|---|
| blank     | No special action (blank itself is not written)       |
| +         | No line feed occurs before the record is written      |
| 0         | A line is skipped before writing the record           |
| 1         | A page skip [0c] is sent before the record is written |

- ACCESS='APPEND' positions the file at the end ready for more writing (implies ACCESS='SEQUENTIAL')

12 Default file names: if the name of an external file is not defined with a "FILE=name" in an OPEN statement Fortran will invent the name "FTnnF001" (nn = unit number). This may be changed by assigning an alias through the operating system, e.g. the system command: "Set FT10F001 My\_file" will force I/O on stream 10 to the file "My\_File" provided that stream 10 has not been OPENed with an explicit name. "UnSet FT10F001" returns to the normal convention.

13 INQUIRE file names: INQUIRE works with full file names: beginning with the filing system and ending with the leaf name. This is what it will give for

```
INQUIRE(UNIT=nn,NAME=TEXT)
```

even if the OPEN gave something much simpler.

When the file is defined with a name rather than a unit it will try to make up a



full file name by doing the following:

- it will first expand any logical name enclosed in chevrons: "<...>"
- if the name starts with "@" it will change "@" for the Currently Selected Directory name (CSD)
- if the name begins with "%" it will change "%" for the current library
- if the name contains "\$" it will try to use the name as it is
- if none of these is satisfied it will add the CSD to the beginning.

14 INQUIRE replies:

when INQUIRE can not determine a variable it returns the following:

```
ACCESS   : 'UNKNOWN'
BLANK    : 'UNKNOWN'
DIRECT   : 'UNKNOWN'
EXIST    : .FALSE.
FORM     : 'UNKNOWN'
FORMATTED : 'UNKNOWN'
NAME     : blank fill
NAMED    : .FALSE.
NEXTREC  : 0 (because the file is not connected or is not direct access)
NUMBER   : 0 (because the file is not connected)
OPENED   : .FALSE.
RECL     : 0 (because the file is not connected or is not direct access)
SEQUENTIAL : 'UNKNOWN'
UNFORMATTED : 'UNKNOWN'
```

15 Error reporting: if the system reports an error it gives a trace-back through the subprogram chain. This is limited by default to a depth of 6 routines. To change the maximum length of this trace-back:

```
CALL TRACEDEPTH(NEWMAX)
```

16 Stack handling: when the program runs out of space it increases its wimp slot to accommodate whatever is immediately needed. The wimp slot is not decreased if the space is no longer needed. You can decrease any unnecessary wimp slot by calling:

```
COMPACT_STACK()
```

It will return the INTEGER number of bytes released if you use it as an INTEGER FUNCTION. This is probably only useful after an initialisation process involving large temporary storage.

[I/O errors in the FDE library](#) reported by IOSTAT=errornumber are listed in the appendix. Those from the standard ([ROOL](#)) Fortlib are also detailed in the appendix ([I/O errors](#)).

These extensions to the ANSI Fortran Standard are for use when reading and writing to files or terminal and VDU screen.

### File names

There is no standard for the allowed length of names in the FILE field of the OPEN statement. This implementation allows names of up to 99 characters after which you will get the message:

*Name truncated in OPEN*

This usually means that you have used a CHARACTER variable with more than 99 characters most of which are blank. In this case it is tidier to use:

```
OPEN(UNIT=iunit, FILE=name(1:LNBLNK(name)), ...)
```

If it is really necessary to have a file name longer than 99 characters you will have to make it an alias, e.g.

```
C      set up an alias for 'name'
      CALL OSCLI('Set My$name '//name)
C      now open the file using alias
      OPEN(UNIT=iunit,FILE='<My$name>',...)
C      note, the '<' and '>' are required
```

### VAX-VMS

VAX Fortran has alternate names for some of the standard I/O functions. The following are accepted in this implementation but cause a compiler warning unless option -zpx1# ('VMS I/O' in FDE) is selected:

| VAX name | Equivalent standard operation            |
|----------|--|
| ACCEPT   | READ (only from terminal, not from file) |
| DECODE   | READ from internal file                  |
| ENCODE   | WRITE to internal file                   |
| TYPE     | PRINT                                    |

The NAMELIST extension to input and output is implemented in the compiler and in the system library; It is a method of grouping variables to read or write them as an entity. It is a complicated extension best understood by reading a VMS manual or consulting the internet.

The implementation in the FDE system library has different limitations from those in the standard Fortlib and conforms more nearly to that used in VMS Fortran 77 (and less so to accepted Fortran 90). The data format has the following requirements:

- The first character on a line is always ignored; useful data start in column 2 or later.
- Multiple entries on a line are allowed, in fact the whole structure could be on one line subject to the line length being less than 512 characters.
- Repeated entries of the form  $r*c$  are allowed; e.g. LIST = 1,2,3\*4,6,7,8
- Sub-indexed assignments such as VECT(3,5) = 10.,20.,30. are not currently supported but you could use VECT =,10.,20.,30./ which would have the same effect but it does require all previous elements of the array to be entered.
- Omitted entries are allowed and data are not transmitted for them:  
e.g. LIST = 1,,,5,6,7,8 will not touch the values in element 2 to 4 of the array.
- The group name and end marker can only be & or \$.

### External File Formats

Files of data are normally stored on disc or electronic media. The characteristics (file name, length etc.) are stored in a header by the filing system (DFS, ADFS, SCSI, SDFS, Fat32FS etc.). You can find these characteristics by clicking with <Menu> over the file icon in the filer window, choosing "File 'name'" then "info".

The files themselves are just a string of 8-bit bytes which are accessed by the FDE System Library following the description you have given in the OPEN statement. The OPEN qualifiers defining how the bytes in the file are to be interpreted are FORM, ACCESS and RECL as described here. (*N.B. these formats are for the FDE system library; they are not the same for the standard Fortlib.*)

**ACCESS=SEQUENTIAL**

- **FORM=FORMATTED** records are just strings of ASCII data, each one being terminated by a 'linefeed' character (0A<sub>hex</sub>). These files are immediately readable by an editor like !Edit and are created with type 'Text' (&FFF).
- **FORM=UNFORMATTED** records each start with a header:  
Two bytes of ASCII (UF) defining an UnFormatted record  
then a 4-byte count of bytes in the record  
followed by the user's data. These files are created with type 'Data' (&FFD).

These are the normal sequential file formats. However, if you have:

**ACCESS=DIRECT**

The data records are formatted with the number of bytes fixed by the compulsory OPEN parameter: RECL=nn where "nn" are the number of bytes of stored data in each record. Both FORMATTED and UNFORMATTED direct access files are introduced with:

two ASCII bytes (DA) followed by four bytes containing the "nn" defined by RECL. The defined record length (RECL) must be big enough for the longest record you wish to store and, for FORMATTED records, must include space for the terminating 'linefeed' character (0A<sub>hex</sub>).

**C to Fortran interface**

Calling C functions from within Fortran programs is particularly important if you are not using the FDE as many of the system functions provided by RISC OS have been written in C. This section explains the way in which the system functions and other user-defined C functions are called from within a Fortran program. It also explains how Fortran functions can be called from within a C program. Many useful system functions can be invoked using the supplied [Application Libraries](#) if you are using the FDE.

**Calling C from Fortran**

A C function can be called from within a Fortran program in the same way that a Fortran function or subroutine is called. Unfortunately the calling conventions for RISC OS C and Fortran are slightly different; Fortran requires that arguments are passed by reference, which means that the Fortran program cannot pass values directly to a C function. Another problem is that C and Fortran handle character strings differently.

**Including the C Libraries**

If you use the standard Fortran system library (Fortlib), it contains the interface to the shared CLIB module so the C library need not be loaded when C functions are linked with a Fortran program. The shared CLIB module must be loaded, as it is for pure Fortran programs using the standard Fortlib and not FDE.

**Differences in Calling Conventions**

Fortran passes arguments to its subroutines and functions using pointers for each of the arguments. This means that a C function which was written solely for use with a Fortran program would take the following form. (Note that the function's name must be written in lower case characters and have an underscore appended if it is to be accessed directly from Fortran.)

```
void cfunc_( long *adi, double *adx)
{
    long i = *adi;
    double x = *adx;
    ...
}
```

This routine could then be called from within the Fortran program by the statement:

```
CALL CFUNC(I,X)
```

C functions which return a value do so in the same way as a Fortran function. For example:

```
int myfunc_(int *adi, float *adx)
{
    int i = *adi;
    float x = *adx;
    int res;
    ...
    return res;
}
```

This function would be called from Fortran with a statement like:

```
J = MYFUNC(I,X)
```

**Differences in String Representations**

In C, a string is a sequence of bytes (characters) terminated by a single byte containing zero (often called the null byte). Fortran character variables are also a sequence of bytes but without the terminating zero. A Fortran routine passes a character variable to a subprogram using the address of the variable and the length as an additional integer argument.

A C function written solely for use with a Fortran program would therefore take the following form:

```
void cfunc_( char *fstring, int flength )
{
    ...
}
```

This C function could then be called from within Fortran by a statement of the form:

```
CALL CFUNC( 'hello world' )
```

in which case it would pass 11 as the second argument. Fortran character variables are rarely null-terminated so the C program must either use the characters carefully, or the string must be copied using `strncpy` to a private buffer. The length arguments are passed immediately after all the other parameters and in the order of their corresponding string address parameters.

### Returning values from C to Fortran

Integer and floating point values from a C function are returned to a Fortran calling routine in the same way as those from a Fortran function.

With CHARACTER valued functions the memory address in which the returned string is to be placed is passed as the first argument to the function and the second argument is the maximum length of the string. Normal arguments follow these.

For example with the Fortran call:

```
CHARACTER*8 FUNCTION FOO
CHARACTER*8 ANSWER
....
ANSWER = FOO(I)
....
```

the C-based function FOO could be written

```
void foo_(char *res, int maxlen, int *i)
{
    ...
    strncpy(res, "The ans", 8);
}
```

Note that CHARACTER\*1 functions are treated in exactly the same way, so an example function to force a character to upper case could be:

```
void toupper_(char *res, int maxlen, char *in,
              int len)
{
    if (len != 1 || maxlen != 1)
    {
        fprintf(stderr, "Illegal call to TOUPPER\n");
        fortran_exit(1);
    }
    *res = toupper(*in);
}
```

The call to `fortran_exit(1)` causes the program to stop with an error code 1 after closing all open files.

### Fortran alternate returns

Alternate returns to a Fortran routine are made with the same code as would be used in Fortran so that

```
RETURN 3
```

in Fortran is the same as

```
return 3;
```

in C.

**Passing function references**

These are also interpreted in the same way by C as they are in Fortran. For example, if the C routine is called with the following Fortran excerpt:

```
EXTERNAL MYFUNC
CALL CPROG(MYFUNC)
```

then the C code for CPROG might be:

```
void cprog_( func )
int (*func)();
{
    ...
    (*func)(...) /* access passed function */
}
```

**Calling Fortran from C**

Fortran functions or subroutines expect their arguments to be passed by reference (i.e. pointers to the data). The lengths of all character string arguments are added to the parameter list after all the others.

When calling Fortran routines from C, use lower case and add an underscore to the end of the routine name

**Accessing Fortran Common Blocks from C**

Common blocks are used within Fortran programs so that data can be made available to one or more of the program units. This section shows how a C module can access a common block which has been declared in a Fortran program unit.

In the following example, the Fortran module declares a common block, called TEST, which is used to store two variables, one of type INTEGER and the other of type REAL. The example C module accesses this common block and prints out the two values.

The Fortran module has the form:

```
C
C This is the Fortran module, f.fcom
C
      PROGRAM FCOMMON
      COMMON /TEST/I,A
      I = 1
      A = 1.0
      CALL CFUNC
      END
```

The C module has the form:

```
/* This is the C module, c.ccom */
#include <stdio.h>

extern struct
{
    int i;
    float a;
} test__;

void cfunc_( void )
{
    fprintf( stderr, "i = %d, a = %f\n", test__.i,
             test__.a);
}
```

The C and Fortran modules can then be compiled and linked from the command line as follows:

```
*cc -c c.ccom  
*f77 -o com f.fcom o.ccom
```

Alternatively use the FDE application to link the c object after compiling the Fortran code.

When the resulting binary file 'com' is executed, the values of I and A will be displayed.

COMMON block names are terminated by two underscore characters, so do not clash with routine names.

The BLANK (unnamed) COMMON area has the name '\_BLNK\_\_' using upper case letters.

More information can be found in the standard Fortran manual, Chapter 13: 'Interfacing other languages with Fortran'.

**ARM Procedure Calling Standard (APCS)**

Both this Fortran and the C compiler conform to the APCS which means it is possible to combine compiled procedures written in both languages and Assembler to make an executable program.

The integer and floating point registers are used for communicating between procedures: integer registers 0 to 3 carry the the first four arguments while the registers 0 carry the returned values of functions (RN 0 for INTEGER or LOGICAL FUNCTIONS and FN 0 for REAL or DOUBLE PRECISION FUNCTIONS).

The integer and floating point registers 0 to 3 are *scratch* registers; their contents do not have to be preserved across sub-program calls. All other registers should be saved and restored. The specific uses of the registers (apart from the program counter and link register which are bound by the machine hardware) do not have defined bindings, however the convention adopted by C and Fortran is shown in the following table:

**General Register bindings**

| Register | Name | Usage  |
|----------|------|--|
| 0        | a1   | first argument, also contains the result of an integer or logical function |
| 1        | a2   | second argument  |
| 2        | a3   | third argument   |
| 3        | a4   | fourth argument  |
| 4        | v1   | preserved register   |
| 5        | v2   | preserved register   |
| 6        | v3   | preserved register   |
| 7        | v4   | preserved register   |
| 8        | v5   | preserved register   |
| 9        | v6   | preserved register   |
| 10       | sl   | stack lower limit  |
| 11       | fp   | frame pointer  |
| 12       | ip   | temporary storage  |
| 13       | sp   | stack pointer  |
| 14       | lr   | link register  |
| 15       | pc   | program counter  |

**Floating Register bindings**

| Register | Name | Usage   |
|----------|------|---|
| 0        | f0   | result of a real or double precision function |
| 1        | f1   | scratch register                              |
| 2        | f2   | scratch register                              |
| 3        | f3   | scratch register                              |
| 4        | f4   | preserved register                            |
| 5        | f5   | preserved register                            |
| 6        | f6   | preserved register                            |
| 7        | f7   | preserved register                            |



**Stack usage**

The stack is used to store ‘frames’ of information on each entry to a sub-program. These can be used to trace back the calls when there is an error and to preserve the values of general registers 4 to 9 and floating registers 4 to 7. The register ‘fp’ is set up to point at the top end of the frame which contains:

**Stack frame**

| Content         | location | content           | location  |
|-----------------|----------|-------------------|-----------|
| saved pc value  | (fp)     | return link value | (fp)-4    |
| saved sp value  | (fp)-8   | saved fp value    | (fp)-12   |
| †saved v6 value | (fp)-16  | †saved v5 value   | (fp)-20   |
| †saved v4 value | (fp)-24  | †saved v3 value   | (fp)-28   |
| †saved v2 value | (fp)-32  | †saved v1 value   | (fp)-36   |
| ‡saved a4 value | (fp)-40  | ‡saved a3 value   | (fp)-44   |
| ‡saved a2 value | (fp)-48  | ‡saved a1 value   | (fp)-52   |
| †saved f7 value | *(fp)-64 | †saved f6 value   | *(fp)-76  |
| †saved f5 value | *(fp)-88 | †saved f4 value   | *(fp)-100 |

Notes:

† These values need only be saved if they are used within the subprogram.

‡ There is no requirement to save these scratch registers though it is often convenient to do so in order to save the arguments to the procedure.

\* The full 12 bytes of each saved floating register must be stored.

**Stack limit**

The stack is filled downwards in memory and the stack limit register ‘sl’ points 256 bytes above the real bottom end of the stack. This is to allow enough room for saving the frame before checking for stack overflow. After storing the frame you should check the limit:

```
CMP    sp,sl                ;check for stack overflow
BLLT   __rt_stkovf_split_small ;extend stack if necessary
```

Where ‘\_\_rt\_stkovf\_split\_small’ is a routine supplied on the library.

If you want use more than 256 bytes of stack you must reserve the space first before using it, so after saving the frame:

```
SUB    ip,sp,#space_needed  ;find new bottom of stack
CMP    ip,sl                ;check for stack overflow
BLLT   __rt_stkovf_split_big ;extend down to ip if necessary
```

Again, ‘\_\_rt\_stkovf\_split\_big’ is a routine supplied on the library.

**Fortran Specific bindings**

Fortran always passes arguments to sub-programs by reference; this means that it is the addresses of the variables which are sent to the sub-program, not the values themselves. This makes for a simple method which allows for information to be passed both ways.

The addresses of the first four arguments are passed in registers a1 to a4. Any additional ones have their addresses stored on the stack in reverse order.

Hence a statement like:

```
CALL SUB1(X1,X2,X3,X4,X5,X6)
```

would be compiled into code like:

```

ADR a1,X1      ;address of X1 into a1
ADR a2,X2      ;etc
ADR a3,X3
ADR a4,X4      ;address of 4th argument in a4
ADR ip,X6
STR ip,[sp,#-4]!;address of X6 on stack first
ADR ip,X5
STR ip,[sp,#-4]!;address of X5 on stack second
BL  sub1_      ;CALL SUB1 (note lower case
               ;& underscore)
ADD sp,sp,#8   ;restore stack on return

```

### Type CHARACTER arguments

Character arguments are also defined by their address, but the actual length of the string is also passed as an extra argument after all the others. N.B. the value of the length is passed, not the address of the length.

If there is more than one CHARACTER argument, the values of their lengths are all passed as extra arguments in the same order as the variables themselves. For example

```

CHARACTER MICHAEL*7,MARY*4
CALL SUB2(MICHAEL,MARY)

```

would be translated like:

```

ADR a1,MICHAEL ;address of Michael
ADR a2,MARY    ;address of Mary
MOV a3,#7      ;length of Michael
MOV a4,#4      ;length of Mary
BL  sub2_      ;CALL SUB2

```

### External references passed as arguments

When a sub-program is called with an argument defining an external reference, the address of this reference is passed. Hence code like:

```

EXTERNAL JAMES
CALL SUB1(JAMES,3.141593)

```

is translated as:

```

IMPORT james_
LDR a1,ext ;load actual address of JAMES
ADR a2,pi  ;address of pi
BL  sub1_  ;CALL SUB1

```

```

...
pi DCF 3.141593
ext DCD james_ ;linker puts JAMES' address here

```

### Alternate RETURN

Alternate returns from a subroutine are simply placed in the register a1, thus:

```

RETURN 3

```

is translated as:

```

MOV a1,#3 ;RETURN number 3
MOV pc,lr ;transfer back control

```

### Results of functions

INTEGER functions return their value in a1 (note, functions can not have alternate returns, so there is no conflict)

REAL and DOUBLE PRECISION functions return their value in f0

LOGICAL functions return 0 or 1 in a1 for .FALSE. or .TRUE.

CHARACTER functions are called with a1 pointing to the answer string and a2 containing the available length; then follow the true arguments.

For example:

```
CHARACTER FUNCC*256,REPLY*256
INTEGER MYCODE
REPLY = FUNCC(MYCODE)
```

would be translated as:

```
ADR a1,REPLY      ;address for answer
MOV a2,#256       ;maximum length for answer
ADR a3,MYCODE     ;address of argument
BL  func_         ;CALL FUNCC
```

COMPLEX functions are called with a1 pointing to the address where the answer is to be stored, then followed by the true arguments

For example:

```
COMPLEX*8 FUNCCX,Z
Z = FUNCCX(X,Y)
```

would be translated as:

```
ADR a1,Z          ;address for answer
ADR a2,X          ;address of first argument
ADR a3,Y          ;address of second argument
BL  funcx_        ;CALL FUNCCX
```

### Writing an Assembler sub-program

Because the Fortran compiler conforms to the ARM Procedure Calling Standard (APCS), assembler programs to be linked with Fortran must also conform and should also conform to the stack usage of Fortran.

By convention, the eight integer and floating registers 0 to 3 are scratch registers, and are not expected to be preserved by a subprogram, whereas all other registers (except the special registers 'ip', 'lr' and 'pc') must be returned in the state they were in when the sub-program was called. Hence the full preamble to a sub-program called SUBNAM is:

```
DCB  "SUBNAM_",0      ;routine name with "_"
                        ;null terminated
ALIGN                    ;pad to word boundary
DCB  &FF,0,0,8        ;255,0,0,name length + pad
EXPORT subnam_
subnam_                 ;executable code starts here
MOV  ip,sp             ;save the stack pointer
                        ;next, save integer registers
STMDB sp!,{a1-a4,v1-v6,fp,ip,lr,pc}
                        ;a1 - a4 need not be saved,
                        ;any of v1 - v6 which are
                        ;going to be used
STFE  f7,[sp,#-12]!    ;store floating registers
STFE  f6,[sp,#-12]!
STFE  f5,[sp,#-12]!
STFE  f4,[sp,#-12]!
SUB   fp,ip,#4         ;fp points to top of frame
CMP   sp,sl            ;check for stack overflow
BLLT  __rt_stkovf_split_small ;extend stack
                        ;use ip to get extra arguments
                        ;your code goes here...
                        ;then at the end:
LDFFE  f4,[sp],#12     ;restore floating registers
LDFFE  f5,[sp],#12
LDFFE  f6,[sp],#12
LDFFE  f7,[sp],#12
                        ;restore integer registers & return
LDMDB fp,{v1-v6,fp,sp,pc}
```

Most assembler code to be used with Fortran will not need to be this complicated; for example, to write

INTEGER FUNCTION IAND(I,J)

which returns the logical AND of I and J needs only four words of executable code:

```
LDR  a1,[a1]      ;get value of I
LDR  a2,[a2]      ;get value of J
AND  a1,a1,a2     ;'and' them into the result
MOV  pc,lr        ;return
```

This code does not set up a new sub-program frame and so will not be recognized in a trace-back. The code including setting up a frame would be:

```
EXPORT  iand_
DCB    "iand_",0   ;routine name
ALIGN
DCB    &FF,0,0,8   ;end of name
iand_  ;start executable code
MOV    ip,sp       ;preamble
STMDB  sp!,{fp,ip,lr,pc};store frame
SUB    fp,ip,#4     ;end preamble
LDR    a1,[a1]     ;get value of I
LDR    a2,[a2]     ;get value of J
AND    a1,a1,a2    ;'and' them into the result
LDMDB  fp,{fp,sp,pc} ;return
```

### Referencing COMMON

The references to labelled common are IMPORTed into a Fortran program with their names in lower case and followed by '\_\_\_' and blank COMMON is referenced by '[\_BLNK\_\_\_]'. (The '[' at the ends are necessary because the assembler objects to names beginning with '\_').

The following assembler excerpt initialises COMMON/MYCOM/ with the numbers 1 to 4 and, when it is called, copies them to blank common:

```
AREA  [_BLNK___],COMMON,NOINIT
%     16      ;4 words in blank common
AREA  mycom___,COMDEF
DCD   1,2,3,4 ;4 words set to 1 to 4
AREA  My_code,CODE
pblk  DCD  [_BLNK___] ;pointer to blank common
pcom  DCD  mycom___   ;pointer to COMMON/MYCOM/
EXPORT trans_
trans_ ;executable code starts
LDR    ip,pcom
LDMIA  ip,{R0-R3}    ;load 4 words from /MYCOM/
LDR    ip,pblk
STMIA  ip,{R0-R3}    ;store them in BLANK COMMON
MOV    pc,lr        ;return
```



# **Debugger tutorial**


There are two well-used methods of debugging your Fortran program

- putting in PRINT statements in strategic places; this can be complicated if your program runs in the WIMP environment
- using the symbolic debugger !DDT

### ***Debugging with PRINT***


In the WIMP environment see [Debugging without !DDT](#), whereas for a stand-alone program, well placed PRINT statements can usually find the problem quickly.

### ***Using !DDT***

To run the symbolic debugger !DDT you will need to obtain the relocatable module 'DDT' and to run without the FDE you will also need the application !DDT 

These are part of the compiler package you have purchased from RISC OS Open Limited ([ROOL](#)).

### **The debug file**

This will have a desktop icon like  which you create by compiling and linking with 'Debug' turned on in the top-right of the FDE compiler options. Either selecting 'Run' in the FDE or double-clicking on this filer icon should start the symbolic debugger provided that the DDT module is loaded (normally done in the !Run file of !FDE).. Click <Menu> over the debug window to select the desired action.

# Appendices



## Compiler Error Messages

List of messages which may appear when compiling a Fortran program (but see appendix A of the [ROOL](#) manual for details of the most recent compiler errors).

### Recoverable Errors

iffy arithmetic shift\n  
 linkage disagreement for \$r - treated as \$m  
 extern \$r mismatches top-level declaration  
 no external declaration in translation unit  
 small (single precision) floating value converted to 0.0  
 small floating point value converted to 0.0  
 operand of # not macro formal parameter  
 ## first or last token in #define body  
 #error encountered \"%s\  
 number missing in #line  
 illegal option -D%s%  
 junk at end of #%s line - ignored  
 unprintable char %#.2x found - ignored  
 duplicate macro formal parameter: '%s'  
 spurious #elif ignored  
 spurious #else ignored  
 spurious #endif ignored  
 assignment to 'const' object \$e  
 differing pointer types: \$s  
 \$s: cast of \$m to differing enum  
 size of a [] array required, treated as [1]  
 sizeof <bit field> illegal - sizeof(int) assumed  
 size of function required - treated as size of pointer  
 size of 'void' required - treated as 1  
 wrong number of parameters to \$e  
 RISC OS (or other) reentrant module has static init. to data \$r

### Warnings

#pragma -b suppresses errors (hence non-ANSI)  
 argument %s of macro %s expanded in %c...%c  
 type disagreement for \$r  
 \$s tag \$b not defined  
 extern clash \$r, \$r clash (ANSI 6 char monospace)  
 label \$r was defined but not used  
 extern 'main' needs to be 'int' function  
 extern \$r not declared in header  
 typedef \$b declared but not used,  
 function \$b declared but not used,  
 variable \$b declared but not used,  
 unused earlier static declaration of \$r  
 implicit return in non-void %s()  
 implicit return in non-void function  
 Missing identifier after #ifdef  
 Unknown directive: #%s  
 Unrecognised #pragma (no '-' or unknown word)  
 Unrecognised #pragma -%c

#ifdef %s may indicate trouble...  
 (possible error): >= %lu lines of macro arguments  
 character sequence %s inside comment  
 Non-ANSI #include <%s>  
 repeated definition of #define macro %s  
 ANSI '%c%c%c' trigraph for '%c' found - was this intended?  
 #define macro '%s' defined but not used  
 \$b is set but never used  
 \$b may be used before being set  
 \$s: cast of \$m to differing enum  
 differing pointer types: \$s  
 wrong number of parameters to \$e  
 '&' unnecessary for function or array \$e  
 Illegal format conversion '%%%c'  
 shift of \$m by %ld undefined in ANSI C  
 division by zero: \$s  
 floating to integral conversion failed  
 \$s: cast between function pointer and non-function object  
 Format requires %ld parameter%s, but %ld given  
 actual type \$t mismatches format '%.\*s'  
 Incomplete format string  
 out-of-bound offset %ld in address  
 lower precision in wider context: \$s  
 implicit narrowing cast: \$s  
 use of \$s in condition context  
 argument and old-style parameter mismatch: \$e  
 explicit cast of pointer to 'int'  
 odd unsigned comparison with 0: \$s  
 'format' arg. to printf/scanf etc. is variable, so cannot be checked  
 ANSI surprise: 'long' \$s 'unsigned' yields 'long'  
 no side effect in void context: \$s

### Warnings from Ansi C library

signed constant overflow: \$s  
 floating point constant overflow: \$s  
 implicit cast from (void \*), C++ forbids

### Warnings from Portable C library

static function \$b not defined - treated as extern  
 missing newline before EOF - inserted  
 differing redefinition of #define macro %s  
 \$s: illegal cast to \$t  
 \$s: cast to non-equal \$t illegal  
 'register' attribute for \$b ignored when address taken  
 differing pointer types: \$s  
 \$s: implicit cast of pointer to non-equal pointer  
 \$s: implicit cast of non-0 int to pointer  
 \$s: implicit cast of pointer to 'int'  
 \$s: implicit cast of \$t to 'int'  
 objects that have been cast are not l-values  
 comparison \$s of pointer and int:\n literal 0 (for == and !=) is only legal case

\$s: cast between function pointer and non-function object

**Errors**

duplicate definition of \$r  
duplicate definition of label \$r - ignored  
duplicate definition of \$s tag \$b  
extern clash \$r, \$r (linker %ld char%s)  
incomplete tentative declaration of \$r  
re-using \$s tag \$b as \$s tag  
type disagreement for \$r  
label \$r has not been set  
Overlarge (single precision) floating point value found  
Overlarge floating point value found  
Missing #endif at EOF  
EOF in comment  
EOF in string escape  
EOF in string  
quote (%c) inserted before newline  
Too few arguments to macro %s(... on line %ld  
No identifier after #if defined  
No ')' after #if defined(...  
Missing identifier after #ifdef  
#include file %c%s%c wouldn't open  
Junk after #include %c%s%c  
Missing '<' or '\ ' after #include  
Too many arguments to macro %s(... on line %ld  
Missing ',' or ')' after #define %s(...  
Missing identifier after #define  
Missing parameter name in #define %s(...  
Missing '%c' in pre-processor command line  
Missing ')' after %s(... on line %ld  
Missing identifier after #undef  
Unknown directive: #%s  
\$s: illegal cast of \$t to pointer  
\$s: illegal cast to \$t  
\$s: cast to non-equal \$t illegal  
bit fields do not have addresses  
Illegal in lvalue: function or array \$e  
Illegal in l-value: 'enum' constant \$b  
Illegal in the context of an l-value: \$s  
illegal in %s: <unknown>  
illegal in %s: non constant \$b  
illegal in %s: \$s  
attempt to apply a non-function  
size of \$c needed but not yet defined  
Illegal types for operands: \$s  
\$c not yet defined - cannot be selected from  
\$c has no \$r field  
'void' values may not be arguments  
illegal indirection on (void \*): '\*\*'

### Fatal Errors

I/O error writing '%s'  
 out of store (for error buffer)  
 out of store (in cc\_alloc)  
 (Compilation of the debugging tables requested with the -g option  
 requires a great deal of memory. Recompiling without -g, with  
 the more restricted -gf option, or with the program broken into  
 smaller pieces, may help.)  
 out of store (in cc\_alloc)  
 Too many errors  
 I/O error on object stream  
 #error encountered '%s'

### *Run time errors*

These can either be arithmetic errors (dividing by zero etc.), addressing errors (e.g. using a bad index to an array) or Input/output errors.

#### **Floating-point exception errors**

These occur when some illegal operation is tried with REAL or COMPLEX numbers. They always give a dump which will appear in a text window for a wimp program. The following example shows what happens when a floating point division by zero is attempted:

```
Floating point exception: division by zero
Postmortem requested
Arg2: 0x0000e600 58880 -> [0x00000000 0x00000000 0x00000001
0x000081a0]
Arg1: 0x0000e5fc 58876 -> [0x3f800000 0x00000000 0x00000000
0x00000001]
80fc in function sub1_
80ac in function MAIN_
9898 in function main
39a29c4 in unknown procedure
```

In this example the error has occurred in the procedure SUB1 at location 80FC(hex). SUB1 has been called with two arguments whose addresses are E5FC and E600(hex) (or 58876 and 58880 in decimal). These addresses are not likely to be much help but inside the square brackets are listed the first four words of the argument (assuming them to be arrays). The reason for this particular failure was that argument 2 was zero. SUB1 was called by the main program at address 80AC(hex). The trace-back actually continues into the calling system routines but this part is not useful. The locations of the procedures in memory can be found in the link map produced with the !FDE 'map' link option or with -map in the link command. A simpler trace-back is produced by the FDE Fortran library.

There are only five kinds of Floating-point exception:

- division by zero: this is obvious
- overflow: the result is too large to be stored in the appropriate word (REAL has a maximum of  $\sim 3.4 \times 10^{38}$  and DOUBLE PRECISION a maximum of  $\sim 1.7 \times 10^{308}$ ).  
Overflow is also reported for an illegal exponentiation like:  
 $-1.0 \times 10^{1.5}$  or  $0. \times 10^{-1.0}$  and for `ATAN2(0.0, 0.0)`
- underflow: the result is too small to be stored; this exception is normally disabled and a result of zero returned. The smallest REAL number which can be stored is  $\sim 1.18 \times 10^{-38}$  and the minimum useful DOUBLE PRECISION number is  $\sim 2.23 \times 10^{-308}$
- inexact: the result is not exactly represented in the word; this exception is also not usually enabled
- invalid operation: this covers a multitude of evils:
  - operations on non-floating point data
  - subtraction of infinities
  - multiplication of infinity by zero
  - division of 0/0 or infinity/infinity
  - MOD(A,B) when A is infinity or B is zero
  - SQRT of negative number
  - overflow on conversion of REAL to INTEGER
  - ACOS or ASIN of number greater than 1
  - SIN, COS, TAN of infinity
  - LOG of zero or negative

These floating point traps may be enabled or disabled using FPSSET from the [Utilities](#) library

### ***Input/Output errors***

#### **I/O errors in the standard library**

These errors may occur when you write to the screen, read from the terminal, or read from or write to a file. They are set by the keyword `IOSTAT=errornumber` in an I/O statement. Below are the error numbers from the standard Fortran Library.

[I/O errors in the FDE library](#) are listed separately.

|       |  |
|-------|--|
| -1    | End of File  |
| 1     | Failed to read a COMPLEX*8                                     |
| 2     | Failed to read a LOGICAL                                       |
| 3     | Unknown read format  |
| 4,5   | repeated sign in reading INTEGER                               |
| 6     | Illegal character in I format input                            |
| 7     | Illegal character in E/F/G format                              |
| 8     | Illegal character in Z format input                            |
| 9     | Attempt to read a formatted record as unformatted              |
| 10    | Write on file after ENDFILE                                    |
| 11-13 | Illegal T editing -- no integer                                |
| 14    | H editing is illegal on input                                  |
| 15    | Apostrophe editing illegal on reading                          |
| 16    | BACKSPACE illegal on sequential files                          |
| 17    | Truncation occurred in ListDirected                            |
| 18    | Line too long in READ  |
| 19-21 | Unexpected + or - sign in format                               |
| 22    | Hollerith field descriptor incomplete at end of line           |
| 23,24 | Quoted string incomplete at end of line                        |
| 25    | Parentheses nested too deeply in format                        |
| 26-29 | Unexpected + or - sign in format                               |
| 30-31 | Unknown character in format                                    |
| 32-37 | Unexpected + or - sign in format                               |
| 38-39 | Hollerith field descriptor incomplete at end of line           |
| 40-43 | Unexpected + or - sign in format                               |
| 44,45 | Parentheses nested too deeply in format                        |
| 46    | Unexpected number, + or - sign in format                       |
| 47    | Quoted string incomplete at end of line                        |
| 48    | Unable to access file for unit %d                              |
| 49    | Unit number out of range                                       |
| 50    | System error, IO transfer without format/direction information |
| 51    | Read on file after ENDFILE                                     |
| 52    | Illegal width in FORMAT  |
| 53    | Failed to read a COMPLEX                                       |
| 54    | Open failure   |
| 55    | Precision greater than width in lw.n or Zw.n format            |
| 56    | Non positive repeat count in list-directed input               |
| 57    | Repeat type is not valid                                       |
| 58    | Illegal character in list-directed input                       |
| 59    | System error, unable to allocate workspace                     |
| 60    | Internal read off end of string                                |
| 61    | Read found end of line   |
| 62    | Internal file full   |

**I/O errors in the FDE library**

[I/O errors](#) in the standard library are listed separately

*In OPEN Statements:*

- 1 Illegal ACCESS code
- 2 Illegal BLANK code
- 3 Illegal FORM code
- 4 Illegal STATUS code
- 5 Illegal unit number
- 6 Can not open OLD file
- 7 Can not open NEW file (e.g. it is a directory)
- 8 Can not open existing file as NEW (or SCRATCH)
- 9 Can not open an interactive unit
- 10 OLD file is not direct access
- 11 Illegal RECL for direct access file

*In CLOSE statements:*

- 12 Unknown ACCESS code
- 13 Illegal unit number

*In REWIND:*

- 14 Direct access REWIND
- 15 File not open

*In ENDFILE:*

- 16 Direct access ENDFILE
- 17 File not open

*In BACKSPACE:*

- 18 Direct access BACKSPACE
- 19 File not open

*While reading/writing:*

- 20 Direct access record number is illegal
- 21 Can not write to unformatted file
- 22 Can not open file
- 23 Illegal unit number
- 24 ACCESS conflict
- 25 Formatted I/O on unformatted file
- 26 Unformatted I/O on formatted file
- 27 Unformatted read header is wrong
- 28 Can not read file
- 29 Unformatted internal I/O not allowed
- 30 Recursive I/O
- 31 List-directed I/O to direct access file
- 32 Buffer overflow on formatted read from file
- 33 Buffer overflow on keyboard read.
- 34 Escape pressed during keyboard input
- 35 No more records reading internal file

- 36 Can not write to formatted file
- 37 No more records writing internal file
- 38 Formatted output buffer overflow
- 39 Data word length incompatible with format descriptor
- 40 Format has no data-transfer descriptor
- 41 Formatted input buffer overflow
- 42 Hollerith/ASCII data illegal in input format
- 43 Formatted input integer overflow
- 44 Tab outside formatted input record
- 45 Negative tab position in formatted output
- 46 Unformatted direct-access output record overflow
- 47 Unformatted input record overflow
- 48 Unexpected ASCII character a
- 49 non-positive tab
- 50 non-positive X-skip
- 51 on-positive Hollerith field
- 52 negative repeat count for data descriptor
- 53 negative repeat count for parenthesis
- 54 B not followed by N or Z
- 55 No count after tab descriptor
- 56 No count (e) in Ew.dEe or Gw.dEe
- 57 d part missing from F,E,D or G descriptor
- 58 w missing from numeric or logical descriptor
- 59 Unexpected right parenthesis
- 60 Too deep parenthesis structure (max. 10)
- 61 Illegal character in binary input
- 62 Illegal character in logical input
- 63 Illegal character in octal input
- 64 Illegal character in hexadecimal input
- 65 Illegal character in integer input
- 66 Illegal character in floating input
- 67 Illegal character in complex input

***In INQUIRE:***

- 101 illegal file name

***Pseudo-errors***

- 3 Finished list-directed input
- 2 Software end-of-file detected
- 1 End-of-file detected



***Glossary of Acronyms***

| <b>Acronym</b>       | <b>Meaning</b>                                     |
|----------------------|--|
| ANSI                 | American National Standards Institute              |
| ARM                  | Advanced (was Acorn) RISC Machine                  |
| ASCII                | American Standard Code for Information Interchange |
| APCS                 | ARM Procedure Calling Standard                     |
| C                    | The 'C' programming language                       |
| DDE                  | Desktop Development Environment                    |
| DDT                  | Desktop Debugging Tool                             |
| FDE                  | Fortran Development Environment                    |
| FORTRAN              | FORmula TRANslation programming language           |
| ISO                  | International Standards Organisation               |
| RISC                 | Reduced Instruction Set Computer                   |
| RISC OS              | Operating System on RISC computers                 |
| <a href="#">ROOL</a> | RISC OS Open Limited                               |

***Comment Form***

Please send comments, complaints, suggestions etc. by email to:

'Fortran@dpmail.co.uk' Remember to say which RISC OS platform you were using, which version of the operating system (RISC OS number) and if relevant, the versions of !FDE, the compiler (f77) and linker (link).

Comments specifically about the compiler and/or linker should be directed to [ROOL](#) at the address given at the end of the compiler manual in the DDE: (Documents.Manuals.FortranPascal/pdf).

Thank you for your help and forbearance.

Fortran Friends.

# Index

## Index

| <b>Index</b>                   |     |
|--------------------------------|-----|
| <b>A</b>                       |     |
| About this Manual              | 9   |
| Accessing libraries            | 23  |
| Acknowledgements               | 2   |
| Appendices                     | 96  |
| Application Libraries          | 31  |
| Available Libraries            | 32  |
| Documentation available        | 32  |
| Naming conventions             | 32  |
| ARM Procedure Calling Standard | 88  |
| Alternate RETURN               | 90  |
| External references            | 90  |
| Floating Register bindings     | 88  |
| Fortran Specific bindings      | 89  |
| General Register bindings      | 88  |
| Results of functions           | 90  |
| Stack frame                    | 89  |
| Stack limit                    | 89  |
| Stack usage                    | 89  |
| Type CHARACTER arguments       | 90  |
| Assembler sub-programs         | 91  |
| Referencing COMMON             | 92  |
| Assumed knowledge              | 9   |
| <b>C</b>                       |     |
| C to Fortran interface         | 84  |
| Accessing Fortran Common       | 86  |
| Calling C from Fortran         | 84  |
| Calling Conventions            | 84  |
| Calling Fortran from C         | 86  |
| Fortran alternate returns      | 85  |
| Including the C Libraries      | 84  |
| Passing function references    | 86  |
| Returning values to Fortran    | 85  |
| String Representations         | 84  |
| 'Check dates' menu item        | 26  |
| command line                   | 68  |
| Comment Form                   | 105 |
| Compiler Error Messages        | 97  |
| Compiler options               |     |
| in FDE                         | 19  |
| on the command line            | 69  |
| Compiling                      |     |

| <b>Index</b>                        |    |
|-------------------------------------|----|
| from the command line               | 68 |
| with FDE                            | 21 |
| Concatenating the help files        | 14 |
| Copyright                           | 2  |
| <b>D</b>                            |    |
| Debugger tutorial                   | 94 |
| Debugging with PRINT                | 95 |
| The debug file                      | 95 |
| Using !DDT                          | 95 |
| Differences from release 2          | 8  |
| Draw Library                        | 33 |
| Calling Sequences                   | 35 |
| Co-ordinate Systems                 | 33 |
| Creating a Draw File                | 33 |
| Errors                              | 34 |
| Naming Conventions                  | 33 |
| Page sizes                          | 33 |
| Routines by subject                 | 33 |
| <b>E</b>                            |    |
| Editing !FDE.!Run                   | 13 |
| Errors in compilation               | 21 |
| Errors while installing and testing | 15 |
| <b>F</b>                            |    |
| FDE                                 | 64 |
| Compiler options                    | 65 |
| Compiling                           | 66 |
| Control window                      | 64 |
| Date checking                       | 64 |
| Directory aliases                   | 68 |
| Directory plan                      | 12 |
| Directory structure                 | 64 |
| Dragging files to                   | 25 |
| Icon Bar menu                       | 64 |
| Libraries                           | 67 |
| Link options                        | 67 |
| Linking                             | 68 |
| Loading files                       | 64 |
| Removing and reordering files       | 68 |
| Run options                         | 68 |
| Running a Fortran program           | 68 |
| Squeeze                             | 68 |

**Index****Index**

|                             |     |
|-----------------------------|-----|
| Throwback                   | 66  |
| Working Directory           | 65  |
| File Formats                | 82  |
| Font Library                | 37  |
| Calling Sequences           | 38  |
| Co-ordinates                | 37  |
| Routines                    | 37  |
| Routines by subject         | 37  |
| Fortran 77 Extensions       | 73  |
| Bit manipulation functions  | 76  |
| Block DO statements         | 75  |
| Character set               | 73  |
| Code-line comments          | 73  |
| Complex*16 arithmetic       | 74  |
| Continuation lines          | 73  |
| Equivalence                 | 76  |
| Hexadecimal Constants       | 73  |
| Hollerith constants         | 73  |
| IMPLICIT NONE               | 74  |
| INCLUDE statement           | 76  |
| Index bounds checking       | 78  |
| Mixed COMMON                | 76  |
| Non-standard assignments    | 74  |
| Other library extensions    | 77  |
| Recursion                   | 77  |
| Symbolic names              | 73  |
| Variable types              | 74  |
| WHILE loops                 | 75  |
| Fortran Desktop Environment | 18  |
| Fortran Programming         | 18  |
| Fortran System Library      | 27  |
| Documentation               | 28  |
| I/O errors                  | 103 |
| <br>G                       |     |
| Getting Started             | 17  |
| Glossary of Acronyms        | 105 |
| Graphics Library            | 39  |
| Calling Sequences           | 40  |
| Co-ordinates, units etc.    | 39  |
| Naming Conventions          | 39  |
| Pixel sizes                 | 39  |
| Routines by subject         | 39  |
| Windows                     | 39  |

|                                |     |
|--------------------------------|-----|
| I                              |     |
| I/O errors                     |     |
| in the FDE library             | 103 |
| in the standard library        | 101 |
| I/O extensions                 | 79  |
| Installation Guide             | 11  |
| Installing 'help' files        | 14  |
| Interactive help               | 19  |
| Introduction                   | 7   |
| <br>L                          |     |
| Library test programs          | 24  |
| Link errors                    | 23  |
| Linker listings                | 25  |
| Linking                        | 22  |
| Linking command line           | 72  |
| Linking libraries              | 23  |
| <br>M                          |     |
| Manipulating FDE files         | 25  |
| <br>N                          |     |
| NAMELIST                       | 82  |
| <br>O                          |     |
| On-line Fortran help           | 22  |
| On-line help from !SrcEdit     | 14  |
| <br>P                          |     |
| Palettes and Pixel Translation | 43  |
| Pragmas                        | 21  |
| Product Software               | 8   |
| <br>R                          |     |
| Reference Section              | 63  |
| Run command line               | 72  |
| Run options                    | 24  |
| Run time errors                | 100 |
| Running a program              | 22  |
| <br>S                          |     |
| Sprite Operations Library      | 43  |
| Calling Sequences              | 44  |

## ***Index***

## **Index**

|                                |     |
|--------------------------------|-----|
| Naming Conventions             | 43  |
| Palettes and Pixel Translation | 43  |
| Parameters                     | 43  |
| Routines by subject            | 43  |
| Sprite names                   | 43  |
| Sprite storage areas           | 43  |
| Squeeze                        | 23  |
| Standard Library               |     |
| I/O errors                     | 101 |
| I/O extensions                 | 79  |
| Symbolic debugger              | 95  |
| System routines                |     |
| by topic                       | 28  |
| Calling Sequences              | 29  |
| <br>T                          |     |
| Testing your system            | 15  |
| Throwback                      | 22  |
| Tutorial on using FDE          | 18  |
| Typeface conventions used      | 10  |
| <br>U                          |     |
| Utilities Library              | 47  |
| Calling Sequences              | 47  |
| <br>W                          |     |
| Wimp Library                   | 49  |
| Co-ordinate systems            | 49  |
| Errors                         | 49  |
| Example Programs               | 50  |
| Library contents               | 49  |
| Parameters                     | 49  |
| Routines you call              | 50  |
| Routines you write             | 53  |
| Wimp program tutorial          | 55  |
| Working directory structure    | 25  |
| Writing a program              | 18  |