

FTN77[®]

User's Guide

Salford —————
————— *Software* ————— *The Compiler Specialists*

IMPORTANT NOTICE

Salford Software Ltd. gives no warranty that all errors have been eliminated from this manual or from the software or programs to which it relates and neither the Company nor any of its employees, contractors or agents nor the authors of this manual give any warranty or representation as to the fitness of such software or any such program for any particular purpose or use or shall be liable for direct, indirect or consequential losses, damages, costs, expenses, claims or fee of any nature or kind resulting from any deficiency defect or error in this manual or such software or programs.

Further, the user of such software and this manual is expected to satisfy himself/herself that he/she is familiar with and has mastered each step described in this manual before the user progresses further.

The information in this document is subject to change without notice.

May 1998

© Salford Software Ltd 1998

All copyright and rights of reproduction are reserved. No part of this document may be reproduced or used in any form or by any means including photocopying, recording, taping or in any storage or retrieval system, nor by graphic, mechanical or electronic means without the prior written consent of the Salford Software Ltd.

Preface

This user's guide describes the facilities available in version 3.55 and later of FTN77(DOS/Win16), the Salford Software Fortran 77 compiler for 80386-, 80486- and Pentium-based Personal Computers running MS-DOS revision 5 and later. This compiler and the applications generated from it can be run under DOS or in a DOS box under Windows 3.1(1) or Windows 95. When used with Salford's ClearWin+, it can also be used to create Win16 applications for Windows 3.1(1) or Windows 95.

This guide also describes the facilities available in version 3.62 and later of FTN77(Win32), the Salford Fortran 77 compiler for 80486 and Pentium based Personal Computers. This edition of the compiler is suitable for the Windows NT Operating System and for Windows 95. It can also be used with ClearWin+ in order to generate Win32 applications for Windows 3.1(1) (using Win32S), Windows 95 and Windows NT.

The guide concentrates on compiler-specific features and those areas of the Fortran language where the ANSI Standard¹ needs amplification.

The guide is not intended to be used as a Fortran language reference manual although chapter 10 does contain a detailed guide to the features of input/output and chapter 12 is a comprehensive guide to character handling. For further information about Fortran 77 the reader is referred to one of the many published texts such as *Effective Fortran 77* by Michael Metcalf (Clarendon Press ISBN 0-19-853709-3).

FTN77 provides a large number of useful subroutines and functions in addition to those specified in the ANSI Standard. Some of the functions that have been provided are defined as intrinsic functions and are described in chapter 11. The remaining functions and all of the subroutines are outlined in chapter 29 and described in the on-line Help systems (one for DOS and one for Windows) and also in a companion volume called the *FTN77 Library Reference* manual.

On the next page you will find a list of chapter headings in this guide. A full table of contents appears after the acknowledgements.

¹ANSI X3.9-1978

Chapter headings in this guide:

page

1. Introduction	1
2. Installation guide and getting started (DOS/Win16).....	5
3. Installation guide and getting started (Win32)	15
4. Compiling with FTN77	21
5. Using /LGO and /LINK.....	39
6. Compiler options	43
7. Using SDBG.....	51
8. Program development	75
9. Optimisation and efficient use of Fortran.....	85
10. Fortran input/output	95
11. Intrinsic functions.....	139
12. Fortran 77 character handling facilities.....	157
13. Language extensions.....	177
14. The in-line assembler	193
15. The in-line assembler and DBOS.....	201
16. Mixed language programming.....	209
17. The COMGEN utility.....	217
18. Calling the Windows API (Win32)	223
19. Using LINK77, RUN77 and Libraries (DOS/Win16)	229
20. SLINK (Win32)	241
21. Using MK and MK32.....	265
22. Using Plato.....	277
23. DBOS (DOS).....	289
24. Running DBOS applications under Windows (Win16)	307
25. Plotter Interfacing (DOS)	313
26. Calling real mode libraries and programs (DOS)	315
27. Execution errors and IOSTAT values.....	325
28. Error and exception handling (Win32)	333
29. Overview of the FTN77 run-time library	335

Some chapters relate only to one version of the compiler (either DOS/Win16 or Win32). These are distinguished in the even page header. In chapters that are largely common to both versions, sections that relate only to one version are presented with a shaded background.

Acknowledgements

* * *

FTN77 is a registered trademarks of Salford Software Ltd.

DBOS, Salford C++, SLINK and ClearWin+ are trademarks of Salford Software Ltd.

FTN90 is a joint trademark of Salford Software Ltd and the Numerical Algorithms Group Ltd.

MS-DOS, Windows, Windows 95 and Windows NT are trademarks of Microsoft Corporation.

BRIEF is a trademark of Borland International Inc.

Intel is a registered trademark of Intel Corporation.

AUTOMAKE is a trademark of Polyhedron Software Ltd.

Table of Contents

1.	Introduction	1
	The compiler.....	1
	High compilation speed.....	1
	Object code.....	1
	Compile-time diagnostics	2
	Run-time diagnostics.....	2
	Source level debugger	2
	In-line assembler.....	2
	Other language extensions.....	3
	Portability aids.....	3
	Mixed language programming and libraries.....	3
	ClearWin+	3
2.	Installation guide and getting started (DOS/Win16)	5
	Hardware requirements	5
	Installing FTN77.....	5
	A simple example	8
	Getting started	9
	HELP!.....	13
	The HELP77 utility	13
3.	Installation guide and getting started (Win32).....	15
	Hardware requirements	15
	Installing FTN77.....	15
	A simple example	16
	Getting started	17
	Using the linker	18
	HELP!.....	19
	Resource compiler (SRC)	19
	How to use this guide to create Win32 applications	20

4.	Compiling with FTN77	21
	The compilation/loading process.....	21
	Compiler source input	22
	Compiler options.....	23
	Compilation listing.....	23
	Compilation messages and statistics	27
	Specifying the properties of the object code	29
	Configuring the FTN77 command.....	32
	Reading compiler options from a file	33
	Compiler directives	34
	The OPTIONS directive	35
	The NOLIST directive	35
	The INCLUDE directive.....	36
	The PROFILE facility.....	36
5.	Using /LGO and /LINK	39
	Load and go	39
	The /LGO option.....	39
	The /LINK compiler option	40
	Relocatable binary libraries and input files.....	41
	The /HARDFAIL option	41
	The /UNDERFLOW option	42
	The /PARAMS option	42
	Opening input/output files.....	42
6.	Compiler options	43
	Quick reference.....	43
	Default compiler options	49
7.	Using SDBG	51
	Introduction.....	51
	Invoking SDBG.....	52
	Location of source files.....	53
	Using SDBG	54
	Desktop window	55
	The stack/status window	55
	Source code window.....	57
	Setting breakpoints.....	58

Setting conditional breakpoints.....	59
Run to line	60
Single stepping.....	60
Examining variables	61
Profiling information	61
Miscellaneous information.....	62
Variables window	62
Data viewing windows	63
Simple expression	63
Array.....	64
Structure	65
Memory dump.....	66
Data view window.....	68
Machine code windows	68
Command line.....	69
Commands	69
Customising the debugger	73
8. Program development	75
Diagnostic facilities.....	75
Compilation diagnostics	75
Linker diagnostics	78
Run-time diagnostics.....	78
Arithmetic overflow checking.....	79
Argument consistency checking	80
Array subscript checking	81
Checking for undefined variables (/UNDEF)	82
ASSIGNED GOTO statement checks.....	83
Character data	83
9. Optimisation and efficient use of Fortran	85
Introduction.....	85
Optimisation	85
The /OPTIMISE compiler option	85
Using a coprocessor	85
Optimisation processes	86
Helping the optimiser	90
Efficient use of Fortran 77	90

	Labels	90
	Intrinsic functions	91
	Statement functions	92
	Common subexpressions	92
	Constants	92
	Dummy array dimensions	92
	Character variables	92
	Format statements	93
	Switching off variable tracking	94
10.	Fortran input/output	95
	Overview	95
	Records	96
	Unformatted record	96
	Formatted record	98
	Endfile record	98
	Files	99
	File existence	99
	File names	99
	File properties	99
	File structure	99
	File position	100
	File access	101
	Internal files	101
	Units	102
	Unit specifier	103
	Internal file identifier	103
	Error and end-of-file conditions	103
	Connecting files	106
	The OPEN statement	106
	User-supplied input/output device drivers	113
	The CLOSE statement	116
	The INQUIRE statement	117
	Data transfer statements	122
	Formatted, sequential access	128
	Unformatted, sequential access	129
	Formatted, direct access	129
	Unformatted, direct access	130

File positioning statements	131
BACKSPACE statement.....	132
ENDFILE statement	132
REWIND statement.....	132
Extensions to the standard.....	132
Extensions to the OPEN Statement.....	133
Extensions to the CLOSE Statement.....	133
Input/output of binary, octal and hex. values	133
Business Editing.....	133
Miscellaneous Input/Output Extensions	137
11. Intrinsic functions.....	139
Introduction.....	139
Non-ANSI intrinsic functions	139
Generic and specific names	140
Intrinsic function names as actual arguments	140
Integer arguments and function results.....	141
Logical arguments and function results	142
Intrinsic function descriptions.....	142
Notes for the table of intrinsic functions	148
12. Fortran 77 character handling facilities.....	157
Character statements	157
Character constants	159
Character expressions.....	159
Character assignments.....	160
Character expressions in parameter statements	161
Character arrays	162
Character substrings.....	162
Data statements involving character entities	163
Input and output of character data.....	164
Comparing character strings	167
Intrinsic functions for handling character data.....	169
Conversion from character to integer and vice-versa.....	169
Length of a character entity.....	170
Locating a substring	170
Portable character comparisons.....	171
Character functions	172

	Characters as dummy and actual arguments	173
	Character entities in common blocks	175
13.	Language extensions	177
	INTEGER and LOGICAL data types	178
	REAL and DOUBLE PRECISION data types.....	179
	Data initialisation in type statements	179
	Hollerith data and ENCODE/DECODE.....	180
	Use of @, \$ and _ characters in names	182
	Long names.....	182
	Octal, hexadecimal and binary values.....	182
	Constants	182
	Input and output	183
	WHILE statement.....	184
	DO WHILE statement	184
	END DO statement	185
	Extra subroutines and intrinsic functions	185
	Internal procedures.....	186
	The INTERNAL PROCEDURE statement	186
	The PROCEDURE statement	186
	The EXIT statement	187
	The INVOKE statement.....	187
	Example of the use of an internal procedure.....	187
	In-line 32-bit assembler.....	188
	Numeric checking of variables and arrays.....	188
	Special form of the DATA statement	189
	Conditional compilation	190
	SPECIAL PARAMETER and /SPARAM.....	190
	CIF, CELSE and CENDIF	190
	IMPLICIT NONE.....	191
	INTERRUPT subroutines	191
14.	The in-line assembler	193
	Introduction.....	193
	The execution environment (Win32).....	193
	The CODE/EDOC facility	193
	Mixing of Intel 32-bit Assembler and Fortran	194
	Labels	195

Referencing Fortran variables.....	195
Literals	196
Halfword and byte forms of instructions	196
Using the coprocessor.....	197
Instruction prefixes.....	197
Other assembler facilities	198
Calling MS-DOS and BIOS	199
Other machine-level facilities	199
Error messages.....	200
15. The in-line assembler and DBOS.....	201
FTN77 programs and the DBOS environment.....	201
Segment selector registers	201
Variable storage	201
Linkage to subroutines.....	203
Trap routines.....	206
The machine code programmer's window	207
16. Mixed language programming	209
Introduction.....	209
Data types	209
Basic data types.....	209
Arrays	209
Character strings	211
Calling FTN77 from C/C++.....	211
Introduction.....	211
CHARACTER variables	211
Arrays	212
INTEGER, LOGICAL and REAL	212
Common blocks.....	212
Calling C/C++ from FTN77 or FTN90.....	212
Calling Windows 3.1 functions	215
Mixing I/O systems in C/C++, FTN77 and FTN90.....	215
17. The COMGEN utility.....	217
Introduction.....	217
Command line.....	217
Source file format.....	217

	Changing the process mode/state	218
	INCLUDE directive.....	218
	Comments.....	218
	Variable declarations.....	219
	Data type mapping	221
	Limitations.....	221
18.	Calling the Windows API (Win32)	223
	Introduction.....	223
	Calling Windows API routines from Fortran.....	223
19.	Using LINK77, RUN77 and Libraries (DOS/Win16).....	229
	Introduction.....	229
	The LINK77 utility.....	230
	LINK77 commands.....	230
	Using LINK77.....	233
	Running the program.....	233
	The RUN77 utility	234
	Libraries	235
	Relocatable binary libraries	235
	Dynamic link libraries	237
	Creating dynamic link libraries.....	238
	Common blocks in dynamic link libraries	239
20.	SLINK (Win32)	241
	Introduction.....	241
	Getting started	241
	Executables.....	245
	Libraries	249
	SLINK command reference.....	253
	Interactive mode.....	253
	Command Line mode.....	259
	Direct import of Dynamic Link Libraries	261
21.	Using MK and MK32.....	265
	Introduction.....	265
	Tutorial.....	266
	Reference.....	270

22.	Using Plato.....	277
	Introduction.....	277
	Getting started.....	277
	The Options Menu.....	278
	The Toolbar.....	278
	Editing Source Files.....	280
	Creating a New File	280
	Open an Existing File.....	281
	Compiling a Single Source File.....	281
	The Message Window	281
	Changing File Options.....	282
	Working with Projects.....	282
	Creating a New Project.....	282
	The Project Window.....	283
	Compiling and Building a Project.....	284
	The Project Menu.....	284
	Projects - Advanced Features.....	284
	Customising Plato	284
	Accelerator Keys.....	285
	Standard Windows	285
	Compiling	286
	Block Marking	287
23.	DBOS (DOS).....	289
	Introduction.....	289
	DBOS and expanded memory managers	290
	Network cards and expanded memory managers.....	293
	DBOS command line arguments.....	293
	Memory management.....	296
	Configuring DBOS.....	297
	The CONFIGDB utility	297
	The DBOS_SET and DBOS_RESET commands.....	298
	The paging algorithm	299
	Writing programs within memory capacity.....	301
	Assembler instructions and the execution environment.....	302
	Using assembler instructions to call DOS and BIOS	304
	DBOS memory map	306

24.	Running DBOS applications under Windows (Win16)	307
	Introduction.....	307
	Installing WDBOS	307
	Windows modes.....	308
	Running programs in a DOS box	309
	Switching back to Windows	310
	WDBOS version number.....	311
25.	Plotter Interfacing (DOS)	313
	The plotter	313
	Cabling requirements	313
	Panel settings.....	313
	Plotting plot files.....	314
26.	Calling real mode libraries and programs (DOS)	315
	Introduction.....	315
	Real and protected modes.....	315
	Rules for calling real mode from protected mode	316
	Calling real-mode libraries	317
	Calling real-mode drivers	323
27.	Execution errors and IOSTAT values	325
28.	Error and exception handling (Win32)	333
29.	Overview of the FTN77 run-time library	335
	Bit-handling	336
	Character-handling.....	336
	Command line parsing.....	337
	Data sorting	337
	Error and exception handling.....	338
	File manipulation	338
	Graphics	340
	Graphics plotter/screen.....	341
	Graphics printer	342
	Hot key	342
	In-line	342

Mouse.....	343
Printer.....	344
Process control	344
Random numbers	344
Real mode.....	345
Sound.....	345
Storage management	346
System information	346
Text screen/keyboard	347
Text windows.....	348
Time and date	348

Introduction

The compiler

FTN77, the Salford Fortran 77 compiler for 32-bit Intel microprocessor systems represents a significant advance for Fortran programmers. It provides fast compilation speed and a range of diagnostic, development and optimisation facilities which together far surpass those usually available on either mainframes, minis or PCs. These facilities and features are summarised below.

High compilation speed

FTN77 achieves typical compilation speeds of between 100,000 and 200,000 lines per minute on a 66mhz Pentium machine.

The linkers, LINK77 and SLINK, are correspondingly fast. This means that the time required to compile and link any reasonably sized Fortran program is considerably less than when using other Fortran compilers and linkers on the same hardware.

Object code

The compiler incorporates many features such as constant folding and common sub-expression recognition which make efficient object programs the norm.

Programs can be compiled in check mode, local optimisation mode (the default), or globally optimised mode.

An option enables you to study the instructions generated in symbolic form.

LINK77 and SLINK can be used for mixed language whilst SLINK will accept any Win32 COFF object file.

Compile-time diagnostics

All compile-time error messages are in plain English and refer to names, labels etc. as appropriate.

Run-time diagnostics

Optional run-time checks are available for array bounds, arithmetic overflow, subroutine argument consistency, undefined variables etc. Full trace-back through subroutine calls, and post-mortem facilities are also available in the event of a run-time error. A language extension allows a check that the numeric value assigned to any numeric variable or array is within a specified range. A compile-time option enables all static variables and arrays to be initialised to zero or to a special “undefined” value at the start of execution. *These checks are sufficient to ensure that almost all faulty programs that contain run-time errors fail cleanly and give an informative error message.*

Source level debugger

The compiler provides a full screen source level debugger which can be used with or without the optional run-time checks being in force. The debugger makes it possible to execute any size of program while viewing the source on screen, with the option to view variable and array values, input/output status, calling sequences etc. in pop-up windows. The debugger is controlled by function keys and commands. The hardware debugging facilities available in the 80486 family of processors are supported and the debugger has a powerful conditional break pointing facility.

In-line assembler

FTN77 supports a CODE/EDOC facility for in-line Intel assembly instructions in 32-bit protected mode.

Other language extensions

Language extensions that are available include `DO / END-DO` and `DO / WHILE` statements, `ENCODE / DECODE`, Hollerith data and business editing. Shifts, masks and address handling are available as intrinsic functions which generate in-line code. Conditional source code compilation is available using `CIF`, `CELSE` and `CENDIF`. An `INCLUDE` directive allows nested source files to be automatically included in a compilation. Declaration and use of variable types have been extended to `INTEGER*1`, `INTEGER*2`, `INTEGER*4`, `REAL*4`, `REAL*8` (synonymous with `DOUBLE PRECISION`), `COMPLEX*8` and `COMPLEX*16`, `LOGICAL*1`, `LOGICAL*2` and `LOGICAL*4`.

Portability aids

The provision of a `/ANSI` compile-time option allows programs to be validated both at compile-time and run-time for compatibility with the ANSI Standard. The `/DREAL` compile-time option allows single precision programs to be automatically compiled using double precision arithmetic throughout (where appropriate). It is not necessary to use the generic intrinsic function names for this feature to function correctly.

Mixed language programming and libraries

Compiled code written in `FTN77` can be linked with code from Salford C/C++ and in both cases the programmer has access to an extensive library functions and subroutines for graphics and other requirements. Also by using the `C_EXTERNAL` statement in Fortran, you can define an interface between routines written in C and Fortran and it is possible to call Fortran routines from C and visa versa.

ClearWin+

ClearWin+ is Salford's revolutionary new Windows programming environment. A simple series of function calls allows you to build a complex Windows interface with little or no knowledge of Windows programming methods. For more information see the Fortran edition of the *ClearWin+ User's Guide*.

2.

Installation guide and getting started (DOS/Win16)

Hardware requirements

The hardware requirements for running FTN77 (DOS/Win16) are as follows:

- An 80386SX-, 80386DX-, 80486SX-, 80486DX- or Pentium-based PC, with a hard disk.
- If you are using an 80386, an 80387 maths coprocessor is recommended. Equally, if you are using an 80486SX, an 80487 coprocessor is recommended (the 80486DX includes its own coprocessor). For those who do not have a coprocessor, a coprocessor emulator is incorporated within DBOS. This offers the ability to run DBOS applications containing floating point arithmetic without the use of a coprocessor, but with unavoidable execution speed penalty. Applications written in Fortran also support a Weitek coprocessor. Further information about the use of coprocessors is given on page 85.
- FTN77 requires version 5 (or later) of MS-DOS, PC-DOS or Novell-DOS.
- The compiler requires a minimum of 2 Megabytes of memory. FTN77 programs are not limited to 640K and are capable of addressing all the memory available.

Installing FTN77

In order to install FTN77 follow these steps. Information given here may be superseded by that given in a README file on the first installation disc.

- Ensure that the following parameters in your **CONFIG.SYS** file have at least the values shown here:

```
FILES=30  
BUFFERS=30
```

- You will have been shipped two high density discs (either 3½ or 5¼ inch). Put the first disc in either drive A or drive B , change to this drive and type:

```
INSTALL
```

As you run through the installation sequence, press **Enter** for the default response and **Esc** to abort from the sequence.

- The **INSTALL** program will read the file **FILES.CFG** from the disk and issue warnings about running other DOS extenders. When you have read the warnings enter “Y” to the prompts.
- **INSTALL** will use the **DOS PATH** to search for a previously installed version of **DBOS**. If this directory cannot be found you will be asked for the name of the directory for **DBOS** which you wish to create. This directory will hold nothing but **DBOS**. You may choose to call it, for example:

```
C:\DBOS.DIR
```

INSTALL will now search for **FTN77**. If it is not found you will be asked for the name of the directory for **FTN77** which you wish to create. This directory could be dedicated to **FTN77** or you might wish to use the same directory as that for **DBOS**. You may choose to call it, for example:

```
C:\FTN77.DIR
```

Now you will be shown the directories and asked to confirm that these are what you want.

- The files are then copied from the release diskettes to the nominated directories, with some of the larger files being converted from the compressed format in which they appear on the release diskette. This process may take several minutes.

When the first diskette has been copied you will be prompted to remove it and insert the second of the two diskettes. The contents of the second diskette are then copied.

- You will then be asked if you want to update your Windows 3 **SYSTEM.INI** file in order to use the **WDBOS.386** virtual device driver. **WDBOS.386** is essential if you wish to run **ClearWin+** applications or run **DBOS** in a DOS box in Windows enhanced mode (further information is provided in chapter 24).
- Next you will be asked if you want your **AUTOEXEC.BAT** file to be updated to include the **DBOS** and **FTN77** directories on the path, and to add the command

DBOS. If you choose the default response “No”, then you will need to edit the AUTOEXEC.BAT file yourself as described below.

- ❑ You will then be given a terminating message, and prompted to terminate. When you terminate the program, you will be reminded to reboot your system.
- ❑ If you have not allowed INSTALL to amend your AUTOEXEC.BAT file, you should now edit the AUTOEXEC.BAT file as follows:
 - 1) Amend the PATH command to include the new directories containing the compiler and DBOS.
 - 2) Add the following command at the bottom of the file:

DBOS

Alternatively, you can type the DBOS command once, before you use the compiler or any other program that uses DBOS. If you are using Windows 3.1(1) this must be loaded before DBOS (see chapter 24) for further information).

If you want to limit the amount of memory which is available to DBOS, you should type:

DBOS <memory limit>

where <memory limit> is the hexadecimal address (for example, DBOS 200000 limits DBOS to two megabytes; for further details see page 305).

After loading DBOS, the HELP77 utility may also be loaded (see page 13).

In the unlikely event that you have difficulty loading DBOS (for example the system hangs, or DBOS crashes with a traceback) you should progressively remove non-essential drivers, TSRs etc. from your CONFIG.SYS and AUTOEXEC.BAT files, in order to find any possible incompatibility.

- ❑ *After you have completed the installation sequence, you must reboot the system.*

DBOS is a Terminate and Stay Resident (TSR) program. Once it has been loaded, you do not need to reload DBOS again unless it is explicitly removed by use of the KILL_DBOS command, or a system reboot is performed. DBOS takes care of the memory management and provides services to FTN77 programs. A full description of DBOS can be found in chapters 23 and 23.

For compatibility with previous versions, FTN77 defaults to /INTS and /LOGS. You may wish to use the FTN77 option /CONFIG at this point to change these options (see page 32).

A simple example

The example below shows a simple Fortran 77 program created using an ASCII text editor and stored in a file named MYPROG.FOR.

```
C      PROGRAM SIMPLE
1      READ *,A,B,C
        IF(A.LT.0.0)STOP
        PRINT *,A,SQRT(A),B*C
        GOTO 1
      END
```

This program can be executed using the FTN77 command:

```
FTN77 MYPROG /LGO
```

with the following typical input/output:

```
[FTN77 Ver x.xx Copyright (c) Salford Software Ltd. 199x]
      NO ERRORS      [<MAIN@>FTN77 - Ver x.xx]
```

```
Program entered
1 2 3

      1.00000  1.00000  6.00000

4 5 6

      4.00000  2.00000  30.0000

-1 0 0
```

Notes:

- ☐ FTN77 assumes by default that source files have the .FOR suffix.
- ☐ A detailed specification of the FTN77 command appears in chapters 6 and 5.
- ☐ FTN77 and all executable files produced by FTN77 need DBOS to be loaded in order to run.
- ☐ In this example, the first parameter is the name of the source file containing the Fortran 77 program and /LGO means “load-and-go” (see chapters 4 to 6 for a full description of all of the options available). Selecting this option means that the program is compiled and executed immediately without the need to use the linker.

Getting started

We have provided a simple statistics program, together with some test data, to show you some of the main features of **FTN77**. The relevant files can be found in the sub-directory **\DEMO** which will have been copied on to your hard disk. To illustrate these features work through the following steps:

- You may find it helpful to print the file **STATS.FOR** (which is about 150 lines long), so that you can refer to it during the demonstration.
- Compile and execute **STATS.FOR** using **FTN77** with the “load-and-go” option, **/LGO**, as follows:

```
FTN77 STATS /LGO
```

The results will be displayed on your screen. Notice that the file suffix “**.FOR**” is implied when compiling (any other suffix should be given explicitly). The **/LGO** option avoids using the linker, and shows how quick it is to compile and execute a program during development.

- Now try some of the features which make **FTN77** an outstanding program development tool. Try the compiler without any debugging aids by typing:

```
FTN77 STATS1 /LGO
```

The program fails, but the co-processor fault is not easily traced in its raw state.

You can now use some of the unique combinations of features provided by **FTN77**, in order to locate the cause of the error. Type:

```
FTN77 STATS1 /CHECK /LGO
```

/CHECK is used so that, when a run-time error occurs, the **FTN77** system can help you to find it. A window appears which contains an error message. Press **Esc** to remove this window and you will see a (red) arrow pointing to the faulty source statement. Press **F1** to display the help window for full details of the debugger facilities. Press **Shift F1** to exit from the debugger.

- In order to illustrate another powerful **FTN77** feature, compile program **STATS2.FOR** by typing:

```
FTN77 STATS2
```

Note that a warning message is displayed saying that the variable “**NVALUE**” is undefined.

Sometimes you will have an undefined variable in a program which is not so easy to locate as this one and the compiler cannot output a compile-time message. In this case, you can use the **/UNDEF** option to pinpoint the undefined variable at run-time. Type:

```
FTN77 STATS2 /UNDEF /LGO
```

The run-time system displays the error message and indicates the faulty source program line. Press **Esc** to clear this window and note the faulty line in the source indicated by a (red) arrow.

- Often you will know where an error is, but not its cause. You can use the **FTN77** window-based debugger to find out. Try the facilities of the debugger with **STATS.FOR** (remember there is no error in this program). Type:

```
FTN77 STATS /BREAK
```

/BREAK implies both **/CHECK** and **/LGO** and causes the system to enter the debugger. A window appears showing the source program, with a (red) arrow pointing to the first statement. Press key **F1**, and the **HELP** window for the debugger will appear. You will see from this that you can step through program execution in one of two ways either:

- statement by statement (pressing **F7**) or by
- setting breakpoints by moving the cursor and pressing **F3**.

You can also:

- display the values of all variables in a routine (**F4**),
- toggle **F5** to switch between the normal display screen for the program (including any graphical output) and the debugger window,
- and print the values of variables and array elements (type: **PRINT <variable>** on the blank command line).

All these and many other facilities are described in the **HELP** window. Try stepping through the source code and printing the values of some variables. When you have finished, press **Shift-F1** to exit from the debugger.

- So far we have always used the **/LGO** option (either explicitly or implicitly) to run the programs. With this option, no intermediate files are created. When the program is free of errors (or when a number of modules must be linked together), it can be compiled with **FTN77** and then linked with **LINK77** as follows. Compile **STATS.FOR** again using:

```
FTN77 STATS
```

This creates the file **STATS.OBJ**. Use the linker to link-load it with:

```
LINK77 STATS.LNK
```

STATS.LNK is a short ASCII file of commands. These commands load **STATS.OBJ** and generate the file **STATS.EXE**. Now run **STATS.EXE** by typing:

STATS

The program will display the same results as before.

(As an alternative to using the information file `STATS.LNK`, the `LINK77` commands can be typed in directly as in the next example.)

- The program `EX1.FOR` illustrates how `FTN77` traps run-time errors which, in large programs, can save many hours of valuable program development time. Print the file `EX1.FOR`. Now compile it, and link-load it as follows:

```
FTN77 EX1 /CHECK
LINK77
LO EX1
FILE
```

(Note that `LINK77` uses a “\$” sign as a command prompt. In this case `LINK77` reports that routine `NOWT` is missing. This is deliberate.)

To execute this program with the option of entering the debugger, use the `RUN77` utility and type:

```
RUN77 EX1
```

From the display, choose the number of one of the deliberate run-time errors. Rerun `EX1.EXE` using the different numbers until you have seen all the errors that can be trapped in this way.

- Now try the profiling facility, which allows you to see how many times each statement in the program has been exercised. Type:

```
FTN77 STATS /PROFILE /BREAK
```

Press **F6** to execute the program, then Down Arrow to trace back to the main program. Press **Esc** to clear the “program terminated” message, then press **F9** to profile the program. Observe that the source window displays the required information. Use the Page Up/Page Dn and arrow keys to scroll the source in the window. Press **Shift-F1** when you have finished with the profile window.

The combination of `/PROFILE` (to ensure that all code has been exercised) with the checking option `/BREAK` makes `FTN77` a very powerful development tool.

- If you want to see the assembler equivalent of the machine code generated by the compiler, type:

```
FTN77 STATS /EXPLIST
```

The compiler produces a file `STATS.LIS` which is a source listing showing assembler interspersed with the Fortran statements.

Using `/LIST` instead of `/EXPLIST` produces a listing file of the source program. Notice how all the compile-time options are listed at the start, and how levels of

nesting of DO and block- IF statements are indicated by .1 .2 etc. following the line numbers at the left hand side.

- You might now like to look at some other FTN77 facilities which are demonstrated by the programs EX2.EXE and EX3.EXE. These programs have already been compiled and link-loaded. Before you execute the programs, you may want to look at the comment lines at the start of the source versions of these programs (EX2.FOR and EX3.FOR).

EX2.EXE can be used to view a file in binary format. The user can scan forward or backward through the file using the cursor keys. This program illustrates a number of FTN77 features: windows, CODE/EDOC, and the use of virtual memory file access.

The program should be used with a filename as a parameter, thus:

```
EX2 MYFILE
```

where MYFILE is any file.

- EX3.EXE, which requires an EGA or VGA colour screen, illustrates the use of a number of low level graphics primitives. The program draws the graphs of some elementary functions. It requires no data and is executed by typing

```
EX3
```

- The programs of the form GRAPHx.FOR also illustrate the use of graphics primitives and require an EGA or VGA colour screen. Some of these (numbers 7 to 11) are supplied in executable form. GRAPH4.FOR requires an on-line graphics printer whilst GRAPH5.FOR requires an on-line plotter. The command lines are, for example:

```
FTN77 GRAPH1 /LGO  
GRAPH7
```

- GWIN.FOR is a substantial program that implements a graphics drawing application and illustrates how pop down graphics menus can be incorporated into an FTN77 program. The program is presented in executable form and is almost entirely mouse driven. In this case you type:

```
GWIN
```

- THREADS.FOR illustrates the multi-threading facilities that are available with FTN77. This program is also supplied in executable form. The command line is:

```
THREADS
```

This brings us to the end of our “Getting started” tutorial. Now that you have been introduced to some of the powerful tools that are available with FTN77, you will be ready to develop your own software.

HELP!

FTN77 provides a **HELP** facility which is invoked, in its simplest form, by typing:

```
FTN77 /HELP
```

This causes the system to output information about the use of **FTN77** on the screen. The **HELP** subsystem consists of a large number of pages of useful information. Each page may be longer than a screen, in which case you may scroll through the text by using the arrow keys and **Page Up**, **Page Dn**.

The following function keys are the most useful:

F1	invokes a brief explanation on how to use the HELP subsystem
F2	returns to the previous HELP screen (which may be from a previous invocation of HELP)
F3	gives an index of HELP pages
Enter	As you move the cursor down the page, references to other pages will change colour. The Enter -key will transfer you to that screen
Esc	leaves the HELP subsystem

Each of the utilities **LINK77**, **RUN77**, **MKLIB77** and the **FTN77** debugger also have a help option.

The on-line **HELP** system includes details of the **FTN77** run-time library. An overview of the subroutines and functions in this library is presented in chapter 29.

The HELP77 utility

HELP77 provides pop up help information from within other applications (**FTN77**-based or otherwise). To use **HELP77** you should include the following statements in your **AUTOEXEC.BAT** file after the **DBOS** command:

```
HOTKEY77  
HELP77
```

Any other hot key programs should be loaded after **HOTKEY77**. After you have rebooted your PC you will find that the key **Ctrl Alt H** will enter this help subsystem at the general index. Items are selected using the cursor keys. The help information provided is the same as that described on page 13 but note that **F2** returns to the previous **HELP** screen even after re-entering the help subsystem.

The **HELP** system includes many fragments of sample code.

3.

Installation guide and getting started (Win32)

Hardware requirements

The hardware requirements for running FTN77 using the Windows 95 or Windows NT operating system are as follows:

- ☐ An 80486DX or Pentium based PC, with a hard disk. It is feasible to use a 386 or a 486SX based PC but these machines are not really adequate for Windows 95 and Windows NT.
- ☐ 16 Megabytes of memory is recommended for the compiler and operating system.
- ☐ 4 to 5 Megabytes of free hard disk space.

Installing FTN77

The compiler and associated tools are distributed on high density floppy disks. The installation program is a Windows application and can be run under Windows NT, Windows 95 or Windows 3.1(1).

If you are installing FTN77 on a network, you should be logged in as the system administrator. You should also install the system whilst running Windows NT or Windows 95. This is necessary to ensure that the “Salford Compiler” group is added to the common areas.

The installation program can be run from either the file manager or the program manager. Insert the first disk into one of the floppy disk drives, say drive A. From either of the above applications, select the **Run...** item from the file menu. A dialog box will appear prompting you for the command line. Enter the command

A: SETUP

and click on the OK button. The installation program will now load from the floppy disk.

Now follow the setup instructions on the screen. These will guide you through the setup procedure. You will be asked to specify the directory where you wish to install the FTN77 compiler. You will also be asked to confirm the location of your Windows SYSTEM directory (normally C:\WINNT\SYSTEM32 for Windows NT). If this is the first installation, you will be asked to enter your name and other related details.

When the installation has been completed, a batch file called FTN77VAR.BAT will have been created in the FTN77 directory. This sets the appropriate environment variables and should be executed before using the compiler. As an alternative, you could add the directory C:\WIN32APP\FTN77 to your path by using the System option from the Control Panel (see the entry for "Environment Variables" in the Windows NT system manual).

A simple example

The example below shows a simple Fortran 77 program created using an ASCII text editor and stored in a file named MYPROG.FOR.

```
C      PROGRAM SIMPLE
1      READ *,A,B,C
        IF(A.LT.0.0) STOP
        PRINT *,A,SQRT(A),B*C
        GOTO 1
      END
```

If you have not already executed FTN77VAR.BAT from the FTN77 directory, you should do so now. This program can be compiled using the FTN77 command:

```
FTN77 MYPROG
```

with the following typical initial output:

```
[FTN77 Ver x.xx Copyright (c) Salford Software Ltd. 199x]
      NO ERRORS      [<MAIN@>FTN77 - Ver x.xx]
```

Notes:

- ☐ FTN77 assumes by default that source files have the .FOR suffix. In this example, the first parameter is the name of the source file containing the Fortran 77 program.
- ☐ A detailed specification of the FTN77 command appears in chapters 4 to 6.

Getting started

We have provided a simple statistics program, together with some test data, to show you some of the main features of **FTN77**. The relevant files can be found in the sub-directory **\DEMO** which will have been copied on to your hard disk. To illustrate these features work through the following steps:

- You may find it helpful to print the file **STATS.FOR** (which is about 150 lines long), so that you can refer to it during the demonstration.
- Compile and execute **STATS.FOR** using **FTN77** as follows:

```
FTN77 STATS
```

followed by:

```
STATS
```

The results will be displayed on your screen. Notice that the file suffix “**.FOR**” is implied when compiling (any other suffix should be given explicitly).

- Now try some of the features which make **FTN77** an outstanding program development tool. Try the compiler without any debugging aids by typing:

```
FTN77 STATS1
```

and run the program.

The program fails, but the co-processor fault is not easily traced in its raw state.

You can now use some of the unique combinations of features provided by **FTN77**, in order to locate the cause of the error. Type:

```
FTN77 STATS1 /CHECK
```

and run the program.

/CHECK is used so that, when a run-time error occurs, the **FTN77** system can help you to find it. When the program is run, a run-time error message is displayed on the console detailing the cause of the error. The line which generates the error can be located by using the debugger supplied with the compiler. A full description of the debugger and its use can be found in chapter 7.

- In order to illustrate another powerful **FTN77** feature, compile program **STATS2.FOR** by typing:

```
FTN77 STATS2
```

Note that a warning message is displayed saying that the variable “**NVALUE**” is undefined.

Sometimes you will have an undefined variable in a program which is not so easy to locate as this one and the compiler cannot output a compile-time message. In

this case, you can use the `/UNDEF` option to pinpoint the undefined variable at run-time. Type:

```
FTN77 STATS2 /UNDEF
```

Now run the program by typing

```
STATS2
```

The run-time system displays the error message and indicates the faulty source program line.

- If you want to see the assembler equivalent of the machine code generated by the compiler, type:

```
FTN77 STATS /EXPLIST
```

The compiler produces a file `STATS.LIS` which is a source listing showing assembler interspersed with the Fortran statements.

Using `/LIST` instead of `/EXPLIST` produces a listing file of the source program. Notice how all the compile-time options are listed at the start, and how levels of nesting of `DO` and block-`IF` statements are indicated by `.1 .2` etc. following the line numbers at the left hand side.

Using the linker

`SLINK` is the Salford 32-bit linker for Win32. It can be employed to link more than one object file by adopting one of three modes of operation that are available.

a) Command line mode

For example typing:

```
SLINK MAIN.OBJ SUB1.OBJ SUB2.OBJ
```

will create an executable called `MAIN.EXE`.

b) Interactive mode

Typing `SLINK` will put the linker in interactive mode and generate a “\$” command prompt. For example:

```
SLINK
$ LOAD MAIN
$ LOAD SUB1
$ LOAD SUB2
$ FILE FIRST
```

generates an executable called **FIRST.EXE**. If the name **FIRST** had not been supplied on the last line then the default name of the executable would be **MAIN.EXE**.

c) **Script file mode**

It is also possible to use an editor to create a script file and then supply this file on the command line. For example suppose the file **SCRIPT.LNK** contains:

```
LOAD MAIN
LOAD SUB1
LOAD SUB2
MAP
FILE FIRST
```

The the command line:

```
SLINK SCRIPT.LNK
```

will produce the executable **FIRST.EXE** together with a link map in the file **FIRST.MAP**. For further information see chapter 20.

HELP!

In order to call upon the **FTN77** help system either

- ☐ use the help option on the compiler command line:

```
FTN77 /HELP
```

- ☐ or click on the help icon in the “Salford Compilers” group,
- ☐ or issue the command

```
WINHELP32 FTN77.HLP
```

from a command prompt.

Resource compiler (SRC)

The Salford resource compiler, **SRC**, is supplied for Windows programmers. To compile a resource type:

```
SRC <RESOURCE_FILE>.RC
```

The resultant object file should be linked to the program using **SLINK**.

For example:

```
FTN77 MYPROG  
SRC MYRES.RC  
SLINK MYPROG.OBJ MYRES.OBJ
```

See the Fortran edition of the *ClearWin+ User's Guide* for further information.

How to use this guide to create Win32 applications

The following chapters are not relevant to Win32:

- ☐ Chapter 19 describing LINK77, RUN77 and libraries.
- ☐ Chapter 23 describing DBOS.
- ☐ Chapters 25 to 26 describing BTRIEVE, connecting to a plotter, and using realmode libraries respectively.

For a details of the Win32 runtime library see chapter 29.

Compiling with FTN77

The compilation/loading process

A Fortran 77 program must be converted to binary form before it can be executed. The process of producing an executable program takes place in two phases.

- **Compilation:** where the Fortran 77 program is checked for syntactic and semantic correctness and relocatable binary code is output to an intermediate file <filename>.OBJ, where <filename>.FOR is the name of the source file.
- **Loading:** using the FTN77 Linker, LINK77 or SLINK, where the relocatable binary code is loaded together with:
 - 1) any other relocatable binary code files,
 - 2) library files that might have been produced by previous compilation(s) with FTN77 (or other compatible compilers, such as Salford C++) and
 - 3) routines from the FTN77 library, other system libraries and dynamic link libraries.

FTN77 is controlled by means of *compiler options* and *compiler directives*. The first part of this chapter describes many of the available compiler options. Some of the compiler directives are described on page 34. A summary of all the compiler options and directives is given in chapter 6.

If the program resides in a single file, the two phases of compilation and loading can be combined by means of the compiler option /LINK whilst the “load and go” option /LGO adds a third phase and automatically runs the program. /LGO and other associated options are described in chapter 5. The use of LINK77 together with RUN77 to link and run DOS/Win16 programs, is described in chapter 19. Details of the Win32 linker SLINK are given in chapter 20.

The compiler option defaults described in this manual are those provided when FTN77 is distributed . Page 32 describes how to reconfigure the compiler to give different defaults.

Compiler source input

The compiler reads programs from text files which have been created by a suitable ASCII text editor.

The source file should be specified as the first parameter to the FTN77 command as follows:

```
FTN77 <pathname>
```

When <pathname> does not include an extension, the compiler searches for the file <pathname>.FOR and if it finds it, it is compiled, otherwise the compiler outputs an error message. Source files must have .FOR as a suffix, or be specified with an explicit extension. Any file name acceptable to the operating system can be used.

For example

```
FTN77 MYPROG
```

compiles the program in the file MYPROG.FOR which is in the current directory whilst

```
FTN77 C:\FTN\PROJECT\MYPROG
```

compiles the file MYPROG.FOR in the directory C:\FTN\PROJECT\.

Only one source file name may be specified unless wildcards are used. For example:

```
FTN77 *.FOR
```

would compile all of the .FOR files in the current directory. In this case an explicit extension (like .FOR) is essential.

Note that any lower case letters in the source file are treated as upper case letters except within character constants or Hollerith data.

Compiler options

Compiler options may be specified as part of the FTN77 command line, for example:

```
FTN77 MYPROG /ANSI /LIST
```

causes FTN77 to compile a program held in a source file MYPROG.FOR using the options /ANSI and /LIST. The options may be abbreviated, but care should be exercised to ensure that the abbreviated form is unique. The subsections below describe some of the options that are available. A complete list is given in chapter 6.

Compilation listing

`/LIST <pathname>` *or* `/LIST`

The `/LIST` compiler option generates a program listing on a given file. If `<pathname>` is omitted, then the default name for the listing file is `<filename>.LIS`.

When a compilation listing is produced, it always contains the following information:

- ☐ Date and time of compilation
- ☐ Source file pathname
- ☐ Compiler version number
- ☐ Compiler options in use
- ☐ Source statement listing
- ☐ Error, warning and comment messages.

The listing consists of all source statements and directives, numbered from line 1. If an `INCLUDE` file is listed (the `INCLUDE` directive is described on page 36), its numbering starts from 1 and numbering reverts to that of the previous file once the `INCLUDE` file has been processed. At the end of the listing of each program unit, three blank lines are output (unless `/PAGETHROW` is also used). Some error messages, warnings and comments are interspersed with the listing of statements and directives; others appear at the end of the listing of a program unit. More details of such messages will be found in chapter 8. A summary of all compiler messages is provided in chapter 6. A number of other features of the listing should be noted:

- ☐ The line numbers of an `INCLUDEd` file are preceded by a slash character, which itself is preceded by two digits specifying the level of nesting of the `INCLUDE` directive.

- The level of nesting (*n*) of DO and/or block-IF statements is indicated by *.n* following the line number for non-zero values of *n*. Each time a DO or block-IF is started, *n* is incremented by one and each time a DO or block-IF is completed, *n* is decremented by one. In the unlikely event that this level exceeds 99, two asterisks appear instead of *n*. *n* is not decremented until the first non-comment line appears following the definition of either the DO label or of the ENDIF statement.
- Non-printing ASCII characters are represented by a query character (?) on the compilation listing. Note that the actual non-printing character is, however, treated as part of the source line to be processed.
- The relative address of each statement is printed in hexadecimal at the right of the line (unless /NO_OFFSET is also used). Relative addresses allow the user to locate the source of run-time errors which occur in parts of the program where no checks have been specified. This is the byte address of the first machine instruction corresponding to the statement, relative to the start of the current program unit. The relative address is incremented for each statement for which the compiler generates code. Code generation ceases for the remainder of the source file when a compilation error is found (unless /PERSIST is also used).
- The information in positions 73 to 80 is separated from the remainder of the line by several spaces. This makes the problem of lines overflowing past column 72 more noticeable from the compilation listing. Note, however, that information contained in positions 73 to 80 is overwritten in the source listing by the address offset information (unless /NO_OFFSET is also used). See also page 27 and the option /NO_WARN73.

/APPEND_LIST <pathname>

is equivalent to /LIST but allows the compilation listing to be appended to the end of an existing file, thus enabling the listings produced by several separate compilations to be sent to the same file. For example, to compile PROG1.FOR and PROG2.FOR and send the resulting compilation listings to the file BOTH.LIS, it is only necessary to type:

```
FTN77 PROG1 /LIST BOTH.LIS
FTN77 PROG2 /APPEND_LIST BOTH.LIS
```

/EXPLIST

is equivalent to /LIST but causes each source statement to be followed by the 32-bit Intel assembler statements into which it was compiled. The assembly language listing is fully symbolic. Information on 32-bit Intel assembler can be found in one of the Intel Programmer's Reference Manuals.

/MAP

implies **/LIST** but also produces a list of all names used in a program unit in a source file (see Figure 4-1) except for system routines and variables that have been declared but not used (see also **/FULLMAP** and **/EXTREFS** in chapter 6). A map contains the following information for each name used in it:

- ☐ **USAGE** local, common, argument etc.
- ☐ **TYPE** integer, real, character etc.
- ☐ **COMMON BLOCK NAME** if appropriate.
- ☐ **OFFSET** The offset field enables the run-time address of a variable to be calculated if desired. For a variable or array in **COMMON**, the offset is its position relative to the start of the common block. For a local variable or array, the offset is its position relative to the local workspace pointer (**EBX%**) or the stack frame pointer (**EBP%**) of the program unit (see chapter 15).
- ☐ **COMMENTS** This information tells the user whether the name has been implicitly typed, has appeared in an equivalence statement, is an array, or (in the case of a local variable) if that variable has never been used in an executable statement in the program unit.

Under Win32, if the **/LINK** option is used together with **/MAP** then a linker map is placed in the file <filename>.MAP.

/XREF

implies **/MAP** and is used to produce a cross-reference listing for each program unit in a source file. The cross-reference listings for a program unit in a source file appear after the compilation listing for that program unit. It excludes variables that have been declared but never used (but see also **/FULLXREF** in chapter 6). An example of a cross-reference listing appears in Figure 4-2. The cross-reference listing contains the source file line numbers of each reference to each label and name in a program unit. The names and labels are sorted into ascending alphanumeric or numeric order respectively.

An asterisk following the line number has a different meaning for a name and for a label:

- ☐ For a name, the asterisk means that the named variable has been modified on the line in question by, for example, an assignment.
- ☐ For a label, the asterisk means that the label was defined on the line in question.

```
SALFORD UNIVERSITY FTN77      VER. x.xx      C:\TESTER.FOR

COMPILER OPTIONS:  LISTING INTS NOCHECK LOGS DYNM OFFSET NOANSI
                   NOPAGETHROW NOSILENT NO_OPTIMISE
```

```

0001          MAP
0002          CHARACTER G(5),NG*72,NAME*14
0003          DIMENSION P(5)
0004          COMMON/ABCD/NG,NAME
0005          DATA P/4.0,3.0,2.0,1.0,0.0/
0006          DATA G/'A','B','C','D','E'/
0007      1      READ(1,4,END=6)NAME,N
0008          READ(1,4)NG
0009          SUM = 0
0010          DO 3 I=1,N
0011.01      DO 2 J=1,5
0012.02  2      IF(NG(I:I).EQ.G(J))SUM = SUM+P(J)
0013.01  3      CONTINUE
0014          WRITE(2,5)NAME,INT(SUM/N+0.5)
0015          GOTO 1
0016      4      FORMAT(A,I5)
0017      5      FORMAT(1X,A,' AVERAGE NUMBER OF POINTS',I4)
0018      6      END

```

Name	Usage	Type	Offset	Comments
G	Local	CHARACTER *1 Array	EBX%+20	Saved
I	Local	INTEGER*2	EBX%-38	Implicitly defined Saved
INT	Intrinsic function	-	-	
J	Local	INTEGER*2	EBX%-40	Implicitly defined Saved
N	Local	INTEGER*2	EBX%-22	Implicitly defined Saved
NAME	Common /ABCD/	CHARACTER *14	+72	
NG	Common /ABCD/	CHARACTER *72	+0	
P	Local	REAL Array	EBX%+0	Saved
SUM	Local	REAL	EBX%-36	Implicitly defined Saved

Common block /ABCD/ is 86 bytes long
End of compilation - Clocked 0.5 seconds

Figure 4.1 A compilation listing and compile-time map

CROSS-REFERENCE MAP

G	2	6*	12	
I	10*	12		
INT	14			
J	11*	12		
N	7*	10	14	
NAME	2	4	7*	14
NG	2	4	8*	12
P	3	5*	12	
SUM	9*	12*	14	
LABEL	7*	15		
1				
LABEL	11	12*		
2				
LABEL	10	13*		
3				
LABEL	7	8	16*	
4				
LABEL	14	17*		
5				
LABEL	7	18*		
6				

Figure 4.2 Cross-reference for the program Figure 4.1

Note: The cross-reference facility has no means of knowing whether the actual arguments of a function or subroutine are modified when the routine is referenced. Thus an asterisk will not appear in the cross-reference listing when a variable is used as an actual argument. If the line number is followed by the character “i”, the number is a line number in an `INCLUDE` file whose name can be found from the source listing.

Compilation messages and statistics

All error, warning and comment messages are output to the compilation listing file if one is specified or implied, otherwise, these messages are output on the screen. Note that these messages are never simultaneously output to the compilation listing file and on the screen. Messages fall into three categories - error, warning and comment.

`/SILENT`

suppresses the printing of warning and comment messages. Unless the `/SILENT` option is in force, the message that is output on the screen at the end of

compilation of a program unit will include the numbers of warning and comment messages.

/IGNORE <n>

allows the suppression of any given compilation error, warning or comment. A typical use of this option might be to permit checking of ANSI-conformity, but to allow the use of Hollerith data. The appropriate error number, 081 in this case, can be found by including the compiler option **/ERROR_NUMBERS**. Thus the compiler could then be invoked with:

```
FTN77 PROG /ANSI /IGNORE 081
```

It is possible to ignore more than one type of error, warning or comment. In this case every error number to be ignored should be preceded by the **/IGNORE** option. *It is important to note that if errors (as opposed to warnings or comments) are ignored, the code generated may be incorrect.*

/DCLVAR

causes the compiler to report an error for each occurrence of a name in a program unit that has not appeared either as an argument, in a type statement, an **EXTERNAL** statement, an **INTRINSIC** statement or a **COMMON** statement. Note that all external function and subroutine names must appear in an **EXTERNAL** statement if **/DCLVAR** is used.

/NO_WARN73

One unfortunate feature of Fortran is that information beyond position 72 of a line is ignored by the compiler. This can often lead to lines being rejected by the compiler with an apparently spurious error message. For example, if a **FORMAT** statement is not continued and yet extends past position 72 the message:

```
**** Unpaired left bracket(s)
```

would be output. In order to make this sort of problem easier to recognise, by default **FTN77** issues a warning when characters are found in columns 73 onwards. This may be tedious if your program uses columns 73 onwards for statement sequence numbers, in which case the default can be changed by using the **/NO_WARN73** option.

/STATISTICS

causes the compiler to output a message on the screen stating the number of lines compiled and the compilation speed.

/ANSI

ensures that all constructs used in a program conform to the ANSI Standard. The **/ANSI** option also informs the **FTN77** run-time system that ANSI-conformity is required. This means that, for example, non-standard use of format descriptors (such as business editing in a run-time format) causes a run-time failure.

/NO_CR

Under DOS/Win16, by default FTN77 expects lines in the source file to be terminated with carriage return, and ignores the following line feed character which is normally present in MS-DOS format text files. The /NO_CR option causes carriage returns in a source file to be ignored, and line feed to be treated as end-of-line. This allows FTN77 to deal with source files in MS-DOS text file format and also source files in UNIX text file format (where end-of-line is indicated simply by line feed). This is particularly useful in a networked environment, with MS-DOS machines connected to UNIX servers.

Specifying the properties of the object code

By default FTN77 always produces relocatable binary code unless the /NO_BINARY option is specified. The relocatable binary code can be either loaded automatically by FTN77 using /LINK or /LGO (see chapter 5) or it can be made available in a (.OBJ) file for loading with the linker, LINK77 or SLINK.

/BINARY <pathname>

specifies <pathname> as the name of the resulting relocatable binary file.

Under DOS/Win16, this is particularly useful when used in conjunction with a wildcard form for the source file name, where the effect is to output all of the code into one file. If the option is omitted, the source file name is used with a .OBJ extension.

/SAVE

By default, local variables are stored *dynamically* on the stack. Space for dynamic variables is reserved on entry to a program unit; the space is freed, and all data values are lost on RETURN. The alternative is to store variables *statically*. Static variables maintain their values until execution terminates. The following types of variables will automatically be static:

- ☐ Any variable that appears in a SAVE statement.
- ☐ Any variable in a program unit containing a blank SAVE statement.
- ☐ Any variables that appear in a DATA statement.
- ☐ Any COMMON block variable.

All other variables will be made static if the /SAVE option is used. However, it is better programming practise to use SAVE statements in program units where static storage is required, rather than relying on the use of a /SAVE compiler option.

Note that while recursion is not part of the Fortran 77 standard, it is supported by FTN77, and in order to obtain different instances of variables for different invocations of a routine called recursively, these variables must be allocated to dynamic storage.

/ZEROISE

is used to set all static variables and arrays (that have not appeared in a **DATA** statement) to zero at the start of execution. It is sometimes found that programs developed with other Fortran compilers, will not run when **FTN77** is used. These problems can often be traced to the assumption that all non-**DATA**ed variables are initially set to zero. *The use of the **/SAVE** and **/ZEROISE** options will often make the program “work”, but efforts should be made to correct the program source by explicitly giving values to the undefined variables.* In this context, the use of the **/UNDEF** option, which causes all non-**DATA**ed variables to be initialised to a known “undefined” value, will be found to be very useful (see below and chapter 8).

Note that it is necessary to compile the *main* program with this option if uninitialised common blocks are to be set appropriately.

/DEBUG

causes **FTN77** to generate symbolic information and to activate the symbolic debugger when fatal errors occur. **/DEBUG** is included in both **/CHECK** and **/UNDEF** which are normally preferred. **/DEBUG** can be used on its own in order to allow the debugger to be used on “dirty” code, which intentionally violates some of the rules of Fortran 77.

/CHECK

implies **/DEBUG** and causes **FTN77** to plant extra object code so that errors (such as array subscript errors and arithmetic overflow) result in a run-time error and entry into the symbolic debugger. **/CHECK** is fully described in chapter 8.

/UNDEF

implies **/CHECK** and also causes **FTN77** to plant code to check that a variable or array element has previously been given a value when it appears in, for example, the right hand side of an assignment statement. **/UNDEF** is fully described in chapter 8.

/INTL and /INTS

With **/INTS**, every variable of type **INTEGER** will become **INTEGER*2** unless it is explicitly declared as **INTEGER*4** or **INTEGER*1**. Similarly every constant of type **INTEGER** will become **INTEGER*2** unless one or more of the following is true:

- ☐ Its value lies outside the range -32768 to +32767.
- ☐ It contains more than five decimal digits including leading zeros.
- ☐ It is not followed by the letter **L** or **B**. For example, 1011L means the **INTEGER*4** constant 1011 and 56B means the **INTEGER*1** constant 56. (This form of constant is not allowed with the **/ANSI** option.)

If the `/INTL` option is used, every variable of type `INTEGER` will become `INTEGER*4` unless it is explicitly declared as `INTEGER*2` or `INTEGER*1`. Also every constant of type integer will become `INTEGER*4` unless it is followed by the letter `S` or `B`. For example, `26S` means `INTEGER*2` constant 26 and `123B` means `INTEGER*1` constant 123. (This form of constant is not allowed with the `/ANSI` option.)

FTN77 only acts as a standard-conforming compiler when `/INTL` is used.

You can use `FTN77 /CONFIG` to determine the default setting (whether `/INTS` or `/INTL`) and to change the default if you prefer.

Note that the use of the `/ANSI` option does not imply `/INTL`. FTN77 provides the intrinsic functions `INTB`, `INTS` and `INTL` for conversion between `INTEGER*1`, `INTEGER*2` and `INTEGER*4` data (see chapter 11).

The DOS/Win16 version of FTN77 is released with `/INTS` as the default whilst the Win32 version is released with `/INTL` as the default.

`/LOGL` and `/LOGS`

With `/LOGS`, every variable of type `LOGICAL` will become `LOGICAL*2` unless it is declared as `LOGICAL*1` or `LOGICAL*4` and every constant of type `LOGICAL` will become `LOGICAL*2`.

If the `/LOGL` option is used, every variable of type `LOGICAL` will become `LOGICAL*4` unless it is declared as `LOGICAL*1` or `LOGICAL*2`. Also, every constant of type `LOGICAL` will become `LOGICAL*4`.

FTN77 only acts as a standard-conforming compiler when `/LOGL` is used.

You can use `FTN77 /CONFIG` to determine the default setting (whether `/LOGS` or `/LOGL`) and to change the default if you prefer.

Note, however, the use of the `/ANSI` option does not imply `/LOGL`. FTN77 provides the intrinsic functions `LOGB`, `LOGS` and `LOGL` for conversion between `LOGICAL*1`, `LOGICAL*2` and `LOGICAL*4` data (see chapter 11).

The DOS/Win16 version of FTN77 is released with `/LOGS` as the default whilst the Win32 version is released with `/LOGL` as the default.

`/DREAL`

specifies that FTN77 should treat all single precision (`REAL`) variables and constants as double precision (`REAL*8`). Correspondingly, all `COMPLEX` variables are treated as `COMPLEX*16`. It is not necessary to use the generic forms of the intrinsic functions when using `/DREAL`. If `/DREAL` is used, the compiler becomes aware of the allowable extensions.

`/DREAL` can make it very easy to compile and execute a single precision program that has produced valid results on a machine with a more accurate single precision

floating point representation, which would otherwise not work satisfactorily without a major conversion.

/OPTIMISE

controls program optimisation and is described on page 85.

/STACK <n>

under Win32, when linking an object module to produce an executable file, the linker needs to know how much memory to set aside for the program stack. *n* is the size of the program stack in bytes.

This option is not generally required as the default supplied by the compiler is more than adequate. For more information see chapter 20.

/WEITEK

under DOS, causes code to be generated for Weitek 1167 and 3167 coprocessors on a 386 and for a Weitek 4167 coprocessor on a 486. If the machine on which the resulting program is run does not have such a coprocessor, the program will fault on the first attempt to perform an operation requiring the coprocessor. A Weitek coprocessor is not required in order to perform the compilation. **/WEITEK** must also appear on the **DBOS** command line before the resulting executable can be run.

Configuring the FTN77 command

The **FTN77** command has many options and it is often convenient to alter the default settings for the options or to create alternative versions of the compiler for specific purposes. For example, you might wish to use the compiler with **/LGO** and **/CHECK** as default options. To configure your compiler type:

```
FTN77 /CONFIG
```

and follow the instructions on the screen. These differ under DOS/Win16 and Win32. Under DOS/Win16 the default options are built into **FTN77.EXE** whilst under Win32 the default options are stored in an ASCII file called **FTN77.OPT** that is automatically called when the compiler is executed.

Under DOS/Win16, you will be presented with a screen of options as they are currently defined (not all options are configurable - only those displayed) and you can select options with the cursor keys and invert them using the space bar. When you have the options as you require them, pressing **Enter** will highlight the pathname of the file to be written with the new options. By default, this is the pathname of the **FTN77** command itself. You can simply press **Enter** again to modify the **FTN77** command itself, or use backspace and other keys to alter the pathname as desired.

When **Enter** is pressed the file is created (if necessary) and set up as an **FTN77** command with the options you have selected.

Notes:

- ☐ The pathname you choose must have the **.EXE** suffix.
- ☐ If at any time you receive a fresh version of **FTN77** you must recreate any configured versions of the **FTN77** command.
- ☐ If at any point you decide to abandon the configuration process press **Esc**.

Under Win32, you are presented with a dialog box with a menu of options. Select **Compiler Options** or **Display Configuration** and click on the **Config** button. Make your changes and then click on the **End** button. From the menu box, click on the **Save** button in order to create a new version of **FTN77.OPT**. It is possible to maintain alternative sets of default configurations by using the compiler option called **/OPTIONS** as below.

Reading compiler options from a file

Compiler options can also be read from a file. The contents of the file can then be used as a set of options that will be combined with the command line. For example, suppose you create a file called **F77.OPT** in the current directory containing the following:

```
/ANSI /LIST
```

The **FTN77** command

```
FTN77 MYPROG /OPTIONS F77.OPT
```

will compile **MYPROG.FOR** using the given options. You can use more than one option file, but an option file must not itself contain the **/OPTIONS** compiler option. Also the options file must not contain options that will be passed to the program using **/PARAMS**.

It is possible to maintain various sets of default configurations by creating a batch file (**F77.BAT** say) of the form:

```
FTN77 %1 /OPTIONS F77.OPT %2 %3 %4 %5 %6 %7 %8 %9
```

for each configuration. You could then create a **.BAT** file for each **.OPT** file in use and your command line would then take the form:

```
F77 MYPROG
```

Compiler directives

FTN77 will successfully process simple programs which contain only Fortran source statements. Such usage of the compiler implies that, for many options such as run-time diagnostics, either the system default will operate or a FTN77 compiler option must be used to change this default for the whole program source. Clearly, in the case of large or complex programs, this is not satisfactory. FTN77 therefore provides a number of compiler *directives* which can be used to give fine control over the facilities that are available.

The following table provides a list of compiler directives that can be inserted into the FTN77 source code. Further information is available via the given cross reference.

Directives	Purpose	Page
OPTIONS	To specify one or more compiler options in a source file	35
NOLIST & LIST	To suppress and re-enable a source listing	35
INCLUDE '<pathname>'	To insert the contents of the specified file at the current point in the compilation. Useful for frequently used common blocks etc..	36
LIBRARY '<pathname>'	Used with /LINK and /LGO to access a specified library.	41
CIF, CELSE, CELSEIF & CENDIF	For conditional compilation	190
IMPLICIT NONE	Forces the programmer to give every variable an explicit type.	191
CODE & EDOC	To start and terminate a 32-bit assembler sequence. These directives are only allowed if an /ANSI compiler option has not been selected	193

The following notes apply to all compiler directives:

- ☐ Each directive must start in column 7 or beyond.
- ☐ Unless otherwise stated, a compiler directive may appear anywhere in a source program.
- ☐ Spaces have no significance except in file names which appear as character constants within quotation marks.
- ☐ Compiler directives other than CODE and EDOC must not be labelled. (CODE and EDOC can be treated as executable statements should this be desired.)

The OPTIONS directive

The **OPTIONS** compiler directive provides a means of specifying many but not all of the compiler options within a source file. For example:

```
OPTIONS ( SILENT, DREAL )
```

could appear in a file instead of using the command line options **/SILENT** and **/DREAL**.

- There are three methods of specifying the current compiler characteristics and their order of precedence is significant. Default compiler options, configured by using the command **FTN77 /CONFIG**, are read first and are superseded by any command line compiler options (including those presented via a **/OPTIONS** file). These in turn are superseded by any compiler directives given in the program.
- An **OPTIONS** directive can only appear before the first program unit, or between program units, in a source file.
- There is no limit to the number of **OPTIONS** directives that can appear in the source file.
- Many of the command line options are available as **OPTIONS** directives. For example: **CHECK**, **DCLVAR**, **DEBUG**, **DOCHECK**, **DREAL**, **EXTREFS**, **FULLCHECK**, **FULLMAP**, **FULLXREF**, **INTL**, **INTS**, **LOGL**, **LOGS**, **MAP**, **UNDEF**, **XREF**, **ZEROISE**.
- Although the compiler option **/LIST** is superfluous in the presence of **/MAP**, **/FULLMAP**, **/XREF**, or **/FULLXREF**, it is nevertheless required if any of the options **MAP**, **FULLMAP**, **XREF** or **FULLXREF** are specified in an **OPTIONS** directive in the program source.

The NOLIST directive

It is possible to suppress all or part of the listing by means of the directives:

```
NOLIST  
LIST
```

which may appear anywhere in the source program or in an **INCLUDE** file. However, note that the **/LIST** option must have been specified on the command line to cause a listing file to be opened. If the compilation listing is suppressed, error, warning and comment messages are output on the screen.

The **NOLIST** directive does not suppress the listing of **INCLUDE** files. The way to do this is described in the next subsection.

The INCLUDE directive

It is possible to include source statements and compiler directives from another file by means of an **INCLUDE** directive which may appear *anywhere* in the current source file. This directive takes one of the forms:

```
INCLUDE '<pathname>'
```

```
INCLUDE '<pathname>', NOLIST
```

where **<pathname>** is the name of a file containing Fortran 77 source statements and compiler directives. If the pathname includes a suffix then this suffix must appear in the directive. For example:

```
INCLUDE 'COLOURS.INS'
```

includes the given file from the current directory.

An alternative form of the **INCLUDE** directive is available, illustrated by the example:

```
INCLUDE <COLOURS.INS>
```

which includes the given file from the Salford directory which under DOS/Win16 is typically C:\DBOS.DIR\INCLUDE whilst under Win32 is typically C:\WIN32APP\SALFORD.

The use of **NOLIST** means that the contents of the **INCLUDEd** file are not output to the compilation listing file. This facility is most often useful when a **COMMON** block is repeated a number of times in a program. Note that a source program may contain any number of **INCLUDE** directives each of which can, optionally, specify a different file. **INCLUDE** directives can be “nested” to a depth of 10. Thus one **INCLUDEd** file can itself contain other **INCLUDE** directives.

The PROFILE facility[†]

It is often useful to know how many times each statement in a program has been executed. Such information may reveal logical errors and can often help in tracing the execution path in the event of a run-time failure. It will also indicate which parts of a program are most heavily used so that those parts can be examined and recoded to improve execution speed should this be considered worthwhile. Profile has a further use in ensuring that test data exercises all parts of a program. The two directives

[†] For Win32, this is available from FTN77 version 2.1 onwards

```
OPTIONS(PROFILE)
OPTIONS(NOPROFILE)
```

or the FTN77 compiler option

```
/PROFILE
```

are used to control the facility. The directives may appear anywhere in the program source. Once profiling is enabled either by option or by directive, each subsequent executable statement is compiled so that a count is kept at run-time of the number of times that statement is executed. The profiling facility is switched off when a `OPTIONS(NOPROFILE)` directive appears in the source file or when the end of the source file is reached.

To obtain a profile listing you should compile and execute your program using the `/BREAK` compiler option together with `/PROFILE` (or the corresponding directives `OPTIONS(PROFILE)` and `OPTIONS(NOPROFILE)`). Run the program - either to a breakpoint (using the cursor keys and **F3**) or to completion (using **F6**). Then press **F9** to obtain profile information on the screen, or issue the command

```
PROFILE <pathname>
```

from the debugger command line to send the information to the specified file. See chapter 7 for further details of using the symbolic debugger.

Note:

- ☐ The `PROFILE` facility can not be applied simultaneously to more than one source file.
- ☐ `/PROFILE` implies `/LIST`. This means that `/PROFILE` generates a listing file and any error messages will appear in the file rather than on the screen. It is therefore better to deal with compile time (syntax) errors before using `/PROFILE`. However, the directive `OPTIONS(PROFILE)` does not depend on `/LIST` in this manner.

5.

Using /LGO and /LINK

Load and go

FTN77 provides a load-and-go facility via the /LGO compiler option so that programs can be quickly compiled, loaded and executed. /LGO can be used with large and complex programs, even those that require the use of libraries. No permanent object or executable file is produced (although there must be enough disk space to accommodate a temporary object file). These features make this facility invaluable for teaching, testing and development where repeated compilations and test runs are the norm.

If you wish to keep a copy of the current executable file then the /LINK option should be used. Under Win32 you can use this together with /LGO.

All the other compiler options summarised in chapter 6 are available (with the exception of /BINARY and /APPEND_BINARY) together with a number of extra options which allow the following:

- ☐ specification of relocatable binary library and input files,
- ☐ underflow trapping,
- ☐ interactive debugging.

The /LGO option

The load-and-go facility is invoked by the /LGO option. For example:

```
FTN77 MYPROG /CHECK /LGO
```

would compile, load and execute the program held in the source file MYPROG.FOR. The order of options on the command line is immaterial, except when an option requires a name, in which case the name must follow it.

The options /BREAK and /DBREAK both imply /LGO[†]. These options also imply either /DEBUG or /CHECK.

These options are summarised in the following table for easy reference.

Option	Debug code planted	Check code planted	Immediate entry to debugger	/LGO implied
/DEBUG	✓			
/CHECK	✓	✓		
/BREAK	✓	✓	✓	✓
/DBREAK	✓		✓	✓

The /LINK compiler option

When the compiler is invoked with the /LINK option, for example

```
FTN77 MYPROG /LINK
```

the linker is automatically invoked after compilation is complete (assuming, of course, that no compilation errors have occurred). The resultant object file is loaded and a corresponding .EXE file is produced. The example above would create a run file called MYPROG.EXE.

If you wish to load other relocatable binary files, in addition to that produced by compilation of the named source file, the /LIBRARY compiler option (or the corresponding directive) should be used (see below).

[†] Under Win32, /BREAK and /DBREAK are available from FTN77 version 2.1 onwards

Relocatable binary libraries and input files

The use of the /LGO and /LINK options is not restricted to programs that require only the FTN77 library. Other system or user relocatable binary (RLB) libraries and RLB input files can be specified by using one or both of the following methods:

- By using the /LIBRARY option in the FTN77 command line. For example:

```
FTN77 MYPROG /LGO /LIBRARY GKSLIB
```

- By using a LIBRARY directive (which must commence at or beyond column 7) in the source file.

```
LIBRARY <pathname>
```

where <pathname> is the name of the file. For example:

```
LIBRARY 'C:\GRAPHICS\GKSLIB'
```

Use of a LIBRARY directive ensures that no RLB library or input file is forgotten when loading a program as the directives are always present in the source file.

If a library filename does not include path information, the current directory is searched, followed by the directory containing the FTN77 compiler.

Notes:

- The compiler will automatically search first for an RLB library or RLB input file with a name suffixed by .OBJ, and then for the unsuffixed filename, even if the library or input filename specified in the FTN77 command does not contain the suffix.
- Under Win32, the option /MKLIB can be used to generate a static library containing a separate COFF object for each function or subroutine (see page 46).
- Dynamic link libraries are not specified on the LIBRARY directive. Under DOS/Win16 they are specified in the LIBRARIES.DIR file, see page 237 for further details. Under Win32, DLLs are normally located either in the directory for the executable or on the PATH.

The /HARDFAIL option

Under DOS/Win16, the use of the /HARDFAIL option causes run time errors to produce a machine level message and return to the operating system, rather than entering the symbolic debugger. This is sometimes useful if the program contains assembler code.

The /UNDERFLOW option

Under DOS/Win16, the use of the /UNDERFLOW option ensures that the first occurrence of underflow in an arithmetical computation is treated as a failure and is not ignored as would otherwise be the case. A large number of occurrences of underflow during execution can result in long execution times because of the way in which the underflow condition is treated. If an underflow is trapped, the message

```
ERROR: Floating point arithmetic underflow
```

is output and the interactive debugger is entered, see chapter 7. If underflows occur during program execution and the /UNDERFLOW option is not used, RUN77 outputs a message at the end of the run specifying the number of underflows that have occurred.

The /PARAMS option

The /PARAMS option is provided to specify command line information for the program. This option is necessary in order to stop FTN77 from scanning the whole command line before the user's program is executed.

For example, suppose that NEWPROG.FOR obtains two filenames FILE1 and FILE2 by means of calls to the system routine CMNAM. These filenames could be specified as follows:

```
FTN77 NEWPROG /LGO /PARAMS FILE1 FILE2
```

An illustrative program appears with the description of the CMNAM routine (see the *FTN77 Library Reference*).

Opening input/output files

Under DOS/Win16, /LGO can be used with a /READ option in the form:

```
/READ <unit> <pathname>
```

This opens the given file for formatted sequential read access on the given unit. /WRITE is similarly used to assign an output file from the command line. /READU and /WRITEU are correspondingly used for unformatted sequential access files. For example:

```
FTN77 MYPROG /LGO /READ 7 MYPROG.DAT
```

Compiler options

Quick reference

Compiler options are specified as part of the FTN77 command line, for example:

```
FTN77 MYPROG /ANSI /LIST
```

Note that options may be abbreviated, but care should be exercised to ensure that the abbreviated form is unique.

The object code produced can be loaded and executed automatically by means of the FTN77 option `/LGO`, see chapter 5 for details. The interactive source level debugging system (see chapter 7) is entered automatically in the event of an error.

There follows a summary of the options available at the time of publication. `/HELP` can be used to obtain an up-to-date summary. Further information is usually available elsewhere in the manual. Please refer to the index for a cross reference.

Some of the options described in this section can be adopted as compiler defaults. The default options can be listed and changed by using the `/CONFIG` compiler option.

`/ANSI`

Checks that the source conforms to the ANSI Standard.

`/APPEND <pathname>`

Synonym for `/APPEND_BINARY` (DOS/Win16 only).

`/APPEND_BINARY <pathname>`

Append the compiler output to the given relocatable binary file. Thus, several compilations can contribute to one binary file (DOS/Win16 only).

`/APPEND_LIST <pathname>`

Append the compilation listing to the given file.

/BINARY <pathname>

Use the given file in place of the default .OBJ file name. Under DOS/Win16, you can use a wildcard form for the pathname.

/BREAK[†]

Implies both /CHECK and /LGO and causes a break to the symbolic debugging facility at the first executable statement.

/BRIEF[†]

Causes all errors, warnings and comments to be output in a form which is compatible with the BRIEF text editor. Programs can then be compiled and then edited whilst still within BRIEF.

/CHECK

Causes code to be planted in order to enable the run-time checking of array bounds, overflow etc., and also to enable the use of the source-level debugging facilities (i.e. /CHECK implies /DEBUG).

/CONFIG

Set-up installation compiler defaults.

/DBREAK[†]

Implies /LGO and causes a break to the symbolic debugging facility at the first executable statement (i.e. like /BREAK, but /CHECK is not implied) (DOS/Win16 only).

/DCLVAR or /DCLVAR <n>

/DCLVAR (or /DCLVAR 2) causes the compiler to report an error for each occurrence of a name in a program unit that has not appeared either as an argument, in a type statement, an EXTERNAL statement, an INTRINSIC statement or a COMMON statement. Note that all external function and subroutine names must appear in an EXTERNAL statement if /DCLVAR is used. /DCLVAR 1 is similar but relaxes the requirement that intrinsics appear in an INTRINSIC statement and that externals appear in an EXTERNAL statement.

/DEBUG

Causes the output of information to allow the use of the source-level debugging facilities (does not imply the run-time checking associated with the /CHECK, /FULLCHECK and /UNDEF options).

/DELOBJ_ON_ERROR

If /LINK or /LGO is used no permanent object module is created. Otherwise, by default an object module will be generated even when errors are present. /DELOBJ_ON_ERROR overrides the default.

[†] For Win32, this is available from FTN77 version 2.1 onwards.

/DOCHECK

Causes a run-time fail if zero-trip DO-loop is executed.

/DREAL

Enables the automatic generation of DOUBLE PRECISION and DOUBLE COMPLEX for all REAL and COMPLEX and intrinsic functions.

/DO1

Causes DO loops to be executed at least once.

/ERRFAIL

Causes a hard fail at the first error encountered in the file.

/ERROR_NUMBERS

Error messages are accompanied by their error number. This number can be used with /IGNORE.

/EXPLIST

Expanded source listing.

/EXTREFS

As for /MAP except that the output is restricted to external references (subroutine, function and common block names).

/FULLCHECK

Implies /CHECK and, in addition, array elements are checked individually.

/FULLDCLVAR

Synonym for /DCLVAR.

/FULLMAP

Implies /MAP but includes unreferenced variables.

/FULLXREF

Implies /XREF and, in addition, includes a cross-reference map of all unreferenced COMMON variables.

/HARDFAIL

Suppresses entry into the debugger in the event of a run time error (DOS/Win16 only).

/HELP

Invoke the window based help system.

/IGNORE <error number>

Causes the compiler to disregard the error whose number follows /IGNORE (note that if errors of severity greater than “warning” are ignored the code produced will probably be invalid)

/IMPLICIT_NONE

Demands that all variables have an explicit type.

/INTL and /INTS

Change the default integer length.

/LGO

Compile, load and execute.

/LIBRARY<name>

Specification of relocatable binary library and input files when using /LGO or /LINK.

/LINK

Compile and load.

/LIST <pathname> or /LIST

Produces a source listing file.

/LOGL and /LOGS

Change the default length for logical values.

/MAP

Compile-time map option (see also /FULLMAP).

/MKLIB <filename.LIB>

Generates a static library containing a separate COFF object for each function or subroutine. This option is useful, when a user wants to link in only a few routines, from a file containing a large number of routines (Win32 only).

/NO_BINARY

Suppresses the creation of an object module.

/NO_COMMENTS

Suppresses comment messages (equivalent to /SILENT 1).

/NO_CR

Treat line feed as end-of-line (rather than carriage return) in source files. Allows direct compilation of either MS-DOS or UNIX-style source files (DOS/Win16 only).

/NO_FAIL

Synonym for /PERSIST.

/NO_FLOATING_TRACKING

Turns off register tracking for floating point values. Register tracking is one of the optimisation processes that is carried out by default (even when the /OPTIMISE option is not used). Register tracking enables a register value to be re-used in preference to a redundant recall of a stored value. Using a register can also increase the precision of this intermediate value.

/NO_OFFSET

Suppresses the output of address offsets on the source listing.

/NO_PEEP_HOLE

Turns off the peep-hole optimiser. This is only effective with the /OPTIMISE option.

/NO_RETYPES

By default FTN77 produces a warning when variables are declared more than once in the same program unit with the same type on each occasion. This option causes this situation to produce an error rather than a warning.

/NO_WARN73

Suppresses warnings for characters appearing in columns 73 and beyond in the source file.

/NO_WARNINGS

Suppresses warning and comment messages (equivalent to /SILENT).

/NOLINK

If /CONFIG has been used to make /LINK the default, /NOLINK restores the default (Win32 only).

/NOTRACKING

Turns off register tracking for all variables (see /NO_FLOATING_TRACKING).

/OLDARRAYS

Allows array subscript checking to be used with array arguments whose corresponding dummy argument is declared with a last subscript of 1. This option is only effective when used with /FULLCHECK.

/ONLY_UNDEF

Implies /UNDEF without /CHECK. It can be useful with programs that do not strictly follow the Fortran rules but want undefined variable checking.

/OPTIMISE

Enables global optimisation. In the default state, local optimisation is in force unless checking is enabled.

/OPTIONS <pathname>

Specifies a file containing additional compiler options.

/PAGETHROW

Causes each program unit in the compilation listing to be printed at the top of a new page.

/PARAMS

Enables the object program to read filenames from the command line.

/PERSIST

By default compilations with errors will terminate as if control break had been pressed. When the command appears in a batch file, the batch process will then be

interrupted. If `/PERSIST` is used, the control break is suppressed and processing of the batch file will continue even when compilation errors have occurred.

`/PROFILE†`

Enables the run-time profile facility.

`/SAVE`

Do not use the stack for storage of local variables and arrays. Otherwise dynamic storage is used for all local variables and arrays. This has the effect of a blank Fortran `SAVE` statement in each subprogram. Its use should normally be avoided.

`/SILENT` or `/SILENT <n>`

Suppress warning and comment messages. `/SILENT 1` suppresses only comments, while `/SILENT 2` (or `/SILENT`) suppresses both comments and warnings. When the `/SILENT` option is not used, the message that is output on the screen at the end of the compilation of a program unit includes the numbers of warning and comment messages.

`/SPARAM <n>`

Used in conjunction with the `CIF`, `CELSE` and `CENDIF` statements to facilitate conditional compilation.

`/STACK <n>`

n is the size of the program stack in bytes used by the linker. The default value is usually more than adequate (Win32 only).

`/STATISTICS`

Print the number of lines compiled and the compilation speed on the screen.

`/UNDEF`

Implies `/CHECK` and also causes code to be planted in order to do run-time checking of any undefined variables or array elements.

`/UNDERFLOW`

Used in conjunction with the load-and-go facility to trap underflow (DOS/Win16 only).

`/UNSAFE`

Used in conjunction with `/OPTIMISE` in order to improve the execution speed of certain programs by using code re-arrangement techniques (see page 89).

`/WEITEK`

Under DOS, causes the compiler to generate code that will use the Weitek 1167, 3167 and 4167 coprocessors. `/WEITEK` is also required on the `DBOS` command line before the resulting executable can be run.

[†] For Win32, this is available from FTN77 version 2.1 onwards.

/XREF

Causes the generation cross-reference listing.

/ZEROISE

All static variables and arrays set to zero at start of execution.

Default compiler options

Many of the above options have a corresponding opposite. For example **/INTL** is the opposite of **/INTS**. If the default setting is changed by using the **/CONFIG** option, then there are occasions when you may wish to use the opposite option in order to temporarily restore the original default. The configuration screen that appears when using **/CONFIG** indicates the name of the opposite when one exists.

7.

Using SDBG

Introduction

In order to improve user efficiency and the usability of Salford products a new set of debuggers, collectively known as **SDBG**, has been designed and implemented. There are three editions in the range:

- ☐ one for **MS-DOS** based applications,
- ☐ one for Windows version 3.1 and above (including Windows for Workgroups and Win16 based Windows 95 applications)
- ☐ and one for Windows NT version 3.1 and above and Win32 based Windows 95 applications.

All three debuggers have been designed to function consistently. The debugger for **MS-DOS** based applications uses a **DOS** screen but emulates a Windows environment. Some of the detail in this chapter, describing this emulation, can be ignored by Win16 and Win32 programmers.

Like other Salford compilers, **FTN77** also incorporates another feature to facilitate debugging, namely the checking options. The checking options, which ensure that a program does not corrupt itself and does not give inconsistent results, are described in chapter 4.

SDBG may be used either:

- ☐ in conjunction with the checking facilities, by compiling with one of the **/CHECK** or **/UNDEF** compiler options, or
- ☐ without the checking facilities by compiling using the **/DEBUG** compiler option.

SDBG allows you to view your source file(s) whilst controlling the execution of your program using function keys and debugger commands. These keys and commands control the following facilities:

- ☐ Program breakpoints
- ☐ Single stepping
- ☐ Display of variables
- ☐ Source and data file inspection
- ☐ Evaluation of expression values
- ☐ Program status display
- ☐ Write/use data breakpoints (using hardware)
- ☐ Machine code debugging
- ☐ Profiling (statement execution count)
- ☐ Input/output stream information
- ☐ Display of the contents of virtual memory
- ☐ Control of screen size

Invoking SDBG

SDBG may be invoked in one of several ways.

- ☐ By compiling a program for immediate execution with the `/BREAK` option, for example:

```
FTN77 MYPROG /BREAK
```

Used in this way `/BREAK` implies the `/CHECK` and `/LGO` options.

- ☐ By compiling the program with the `/DBREAK` option. This option is similar to the `/BREAK` option except that it does not imply the `/CHECK` option which causes the compiler to plant checking code.
- ☐ By linking together one or more `.OBJ` files produced with `/CHECK` or `/DEBUG` options and executing the resultant `.EXE` file as follows.

For DOS programs under `DBOS` use `LINK77` and then type,

```
RUN77 MYPROG /BREAK
```

For Win16 executables use `LINK77` and then type,

```
WINDBG MYPROG
```

For Win32 executables use `SLINK` and then type,

```
SDBG MYPROG
```

The source file for each section of the code to be debugged should be available exactly as it was compiled (i.e. you must not edit these source files prior to using the debugger).

Assuming that no compile-time error is encountered, each of these commands will cause your program to be suspended at the first executable statement in a module compiled with /DEBUG.

The /DEBUG and /DBREAK options cause the compiler to plant sufficient information to enable SDBG to operate, but specify that no checking code is to be planted.

In general, it is better to debug a program compiled with checks, but /DBREAK and /DEBUG are very useful in the following cases:

- When a problem does not manifest itself when the checks are enabled. Often this is a consequence of a calculation being performed with undefined variables or array elements and you are advised to compile the program using the /UNDEF option before using SDBG.
- When the program is just too large to fit in memory when compiled with checks.
- When the program runs too slowly when compiled with checks.

Programs which are not checked may well overwrite themselves and/or the tables which SDBG uses to interpret their behaviour. This can produce unpredictable results.

Location of source files

By default the debugger will look for the source files in the directories they occupied at compile time. If the source files have been moved, there are two methods for specifying alternative directories to for the debugger to search.

Firstly you can specify the environment variable SOURCEPATH in your AUTOEXEC.BAT or its equivalent. This can contain a list of paths. Semicolons are used to separate the paths in the same way as the standard PATH variable. For example:

```
SET SOURCEPATH=C:\COMMON\SOURCE;C:\USERS\PROJ;W:\SRC
```

Secondly, the Windows debuggers can take an optional command line parameter that specifies a source path. This path will replace any path brought in by the SOURCEPATH environment variable. For example

```
SDBG SIMUL /SOURCEPATH  
C:\COMMON\SOURCE;C:\USERS\PROJ;W:\SRC
```

/SOURCEPATH can be abbreviated to /SP.

Using SDBG

The first task SDBG will carry out is to save the running program's screen display and replace it with the debugger screen, switching to text mode if required. SDBG only displays information in text mode although you can debug programs that use graphics modes supported by the Salford graphics library.

SDBG makes use of a windowed interface. In common with other user interfaces a mouse is not absolutely necessary but is extremely useful. The mouse cursor will appear as a one character block in the middle of the screen.

The window that appears on top of all the others is called the *current* window. The current window will respond to any key-presses or mouse actions. It can be distinguished by the double line border surrounding the window. All other windows have a single line border. You can change the current window by pointing at another window and pressing and then releasing the left mouse button. In this case the window you pointed to will be brought to the front and you will see the border change.

You can cycle through the currently open windows by pressing Alt+N. The current window can be moved by pointing to its title bar and pressing the left mouse button. While the button is depressed you can 'drag' the window to its new location.

At the bottom right hand corner of a window you will see that the border thins from a double line to a single line. The single line denotes the fact that you can resize the window. This is achieved by moving the mouse to this area and pressing and holding the left mouse button. You can then drag the window corner to its new size.

At the top left corner of most windows you will see a box character (shown as [■]). By moving the mouse over this area and pressing the left mouse button the window will close. You can also close a window by pressing Alt+F4. Some windows will close when the Esc key is pressed.

When the SDBG screen initially appears it will contain three windows that sit on top of the so called *desktop* window. Namely

- a *stack/status* window,
- a *source code* window,
- a *variables* window.

If SDBG was invoked because of a run-time error, a description of the problem is displayed in the stack/status window. Otherwise the stack/status window will initially be hidden behind the source window which will show the current execution point.

Other windows called *data view* windows can be opened by the user when required. These five differing types of window are described in the following sections.

Desktop window

All visible windows sit on top of the desktop. This is a blue and white hatch with the bottom line displaying help and status information. The status line is mostly made up of a line of white text on a blue background. This gives a list of the most common key presses for the current window. This status line is sensitive to the **Alt** and **Ctrl** keys being depressed. You can also click the left mouse button over a key description and the key press will be simulated. The rightmost six characters show the current debugger mode. The alternatives are:

Status	Meaning
PAUSE	The program has stopped because of a breakpoint and is awaiting commands.
STOP	The program has stopped because of a runtime error (which will be displayed in the stack/status window). You will not be able to continue, step or run the program from this point.
END	The program has terminated.
RUN	The program is running and can be paused by pressing Ctrl+Break . SDBG does not automatically switch to the program screen because most switches would be unnecessary and waste time. The screen is switched as required.

The stack/status window

The stack/status window can be brought to the top by pressing **Alt+C** in any other window. The stack/status window provides two uses:

- ☐ to display the reason why SDBG has been entered.
- ☐ to display the current call stack,

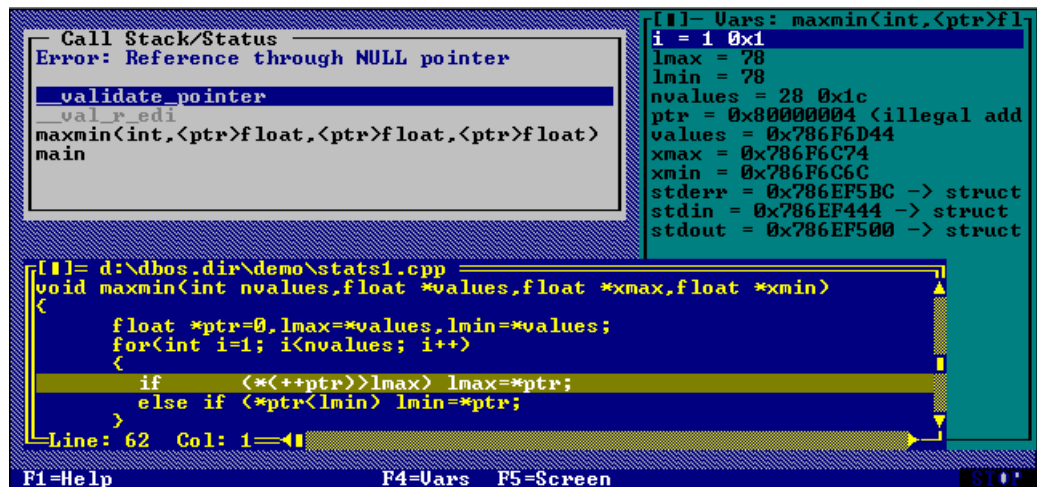


Figure 1. An screen shot from the DOS version of the debugger.

The status part of this window gives the reason that SDBG has been entered.

The stack part gives a trace back through the active call stack. This is a list of the active subroutines and functions. Routines that have debugging information, and therefore can be debugged at the source code level, are displayed in black text. Routines having no debugging information are shown in grey text. You can view the source code and variables for any routine in the call stack with debugging information by either:

- ☐ moving the bar in the status window to the line containing the routine name and pressing **Enter**,
- ☐ or by double-clicking the left mouse button over the routine name.

If you try to open a routine with no debugging information a machine code window will appear. This may appear confusing if you are not familiar with programming at this low level. If one appears, simply click the close button or press **Alt+F4** and it will disappear.

The stack/status window does not have a close button and cannot be closed by pressing **Alt+F4**. This is because the current call stack and status are always relevant.

There are some key presses that apply to every window.

They are as follows:

Key	Action
F1	Help
Alt+F4	Close window

F5	Display user screen
F6	Run or continue the program
F7	Single step the program
F8	Single step over
Alt+C	Display call stack window
Alt+N	Next window
Alt+X	Exit SDBG

Source code window

At a basic level a source window is just a window that shows the contents of a source file. When SDBG is first entered the source window will display the source code for the current execution point. A red bar denotes the first line that caused SDBG to be entered. You can display the source code for any routine in the call stack (assuming the routine was compiled with debugging information) by selecting the routine from the call stack window as described above. Each routine is displayed in a separate window. The routines that are not at the top of the call stack will have their execution point marked with a brown bar.

You can move around in a source window in a manner that is very similar to a text editor or word processor. However, the *text cannot be changed*. The current position is marked by a cursor, which will initially be on the same line as the execution bar. You can move the cursor around the source window with the mouse or using the keyboard.

The right most edge of the source code window contains a scroll bar. You can move this either by clicking the left mouse button whilst the mouse cursor is in the scroll bar or by dragging the scroll bar (you drag the scroll bar by pressing the left mouse button and moving the mouse whilst keeping the left mouse button pressed). To move the source code up or down one line at a time, click on the arrows at the top and bottom of the scroll bar. The source window now also contains a horizontal scroll bar.

You can also use the following key presses to navigate the window:

Key	Meaning
Left arrow	Left one character
Right arrow	Right one character
Up arrow	Up one line

Down arrow	Down one Line
Page Up	Up one page
Page Down	Down one page
Ctrl+Home	Start of file
Ctrl+End	End of file
Ctrl+Page Up	Move up the call stack
Ctrl+Page Down	Move down the call stack
Home	Start of current line
End	End of current line
Ctrl+O	Go to instruction point (Origin)
Ctrl+G	Go to line number
Ctrl+S	Search for text (case insensitive)
Ctrl+A	Search for text specified by the last Ctrl+S command

The Win16 and Win32 debuggers have buttons on the toolbar to move up and down the call stack. They also have a bookmark facility. This is accessed by selecting Bookmarks from the Window menu. Bookmarks can be set, used or deleted.

The most common actions performed on source code in a debugger are usually 1) setting breakpoints, 2) single stepping and 3) running the program. Several key strokes are available to help you do this.

These are summarised in the table below:

Key	Meaning
F2	Set or reset breakpoint
Shift+F2	Set or reset a conditional breakpoint
F3	Get to current line
F6	Run program
F7	Step to next source line
F8	Step to next source line and step over any routine calls

Setting breakpoints

The F2 key acts as a toggle. It will set a breakpoint on a line that has no existing break point. Alternatively, it will remove a breakpoint if one already exists on the highlighted line. This only works if the line is an executable statement. So pressing

F2 when the cursor is over a comment line will have no effect. It will also have no effect when the cursor is over a declaration. Breakpoints are marked by a white bar. Once you have set the breakpoints required you can continue the program by pressing F6. A message box appears if the line is can not be used as a breakpoint.

You should take care when using breakpoints. If the line of code is never executed, due to an IF condition, the program will not stop.

Setting conditional breakpoints

In most circumstances programs can be successfully debugged by setting breakpoints, running the program and examining data. There are some problems which are difficult to debug using simple breakpoints. For example an iterative loop which goes wrong on the 1563th iteration would be very time consuming to debug. You could add extra code to the program to allow SDBG to activate a breakpoint on the 1562nd loop. However, a quicker method is to use a conditional breakpoint. A conditional breakpoint is one which only activates when a given condition is satisfied.

A conditional breakpoint is formed in three parts. Firstly there is an initial delay. This is the number of times the breakpoint has to be executed before it will activate. Secondly there is a repeated delay. This allows you to activate a breakpoint at predetermined intervals. Thirdly, there is an optional expression. When a breakpoint is about to activate, the expression will be evaluated. The breakpoint will only activate if the result of the expression is non-zero.

When you press Shift+F2 an input box will be displayed. You should type into this box the number of times the breakpoint can be executed before it is activated. Entering '5' will cause the breakpoint to activated the 5th time this line is executed. You will then be asked for the number of executions between subsequent breakpoints. You will finally be given an input box into which you can type an expression that will control whether a breakpoint activates or not. You can leave this expression blank if it is not required.

The following table gives examples for the three settings:

Requirement	Initial number	Repeat number	Expression
Stop after the 198th iteration	198	1	
Stop after the 7th iteration and every subsequent 11th	7	11	
Stop when <i>eps</i> is greater than 1	1	1	<i>eps</i> >1
Stop when <i>eps</i> is greater than 1 and I know this is after the 654th iteration	654	1	<i>eps</i> >1

You should note that there is a speed penalty if SDBG has to calculate the result of an expression. Indeed a small speed penalty will result from setting any breakpoint. This is in direct proportion to the number of breakpoints encountered.

You can set the initial delay to zero. This implies that the breakpoint will never activate which can be useful when trying to establish how many times a certain point is reached. You can cancel a conditional breakpoint by moving the cursor to the line and pressing either F2 or Shift+F2. In fact the standard breakpoint is a conditional breakpoint with the delays set to one. You can display the status of all system breakpoints with the 'breakpoints' command (see page 72).

Run to line

One important variation on the 'set breakpoints and run' idea is that of 'get to here'. This is achieved by placing the cursor on the line you would like the breakpoint to appear and pressing F3. SDBG will set a temporary breakpoint at that line, run the program and then reset the breakpoint. This works in a similar manner to the key sequence F2, F6, F2. Again you should be aware that your program may not stop if the code is never executed due to say an IF statement.

Single stepping

Single stepping offers an alternative to setting breakpoints. It allows you to trace the flow of execution a single line at a time. There are two possible methods of single stepping, *step into* and *step over*.

The first method (*step into*) will enter any function or subroutine which contains debugging information. This is useful when you want to follow the logical flow through several routines. You can use the *step into* method by pressing F7. If you step into a new subroutine or function a new source window will be opened. This will display the code for the new subroutine/function. Alternatively, if you execute a return statement, the current source window will be closed and the window with the calling line will be made the current window.

The second method (*step over*) will execute the current line but will not enter a subroutine or function even if debugging information is present. This latter case is useful when you are sure that a subroutine or function is working correctly and you do not want to trace the call through the routine. Whilst a new window will never be opened with the *step over* method, it is possible for the current source window to close due to the execution of a return statement. The *step over* method is performed by pressing F8.

Examining variables

The simplest method for examining any of the current active variables is to use the *variables* window. This window presents a list of all variables that are accessible from the current scope. This list is sorted into scope order and then ascending alphabetical order. The variables window is made the current window by pressing F4.

If you want to examine a particular variable, you can do this by opening a *data view* window. Once opened, a data view window will remain open until the variable goes out of scope or you choose to close it. The contents of the window will be updated each time a break point (or single step action) is encountered.

The source window provides the user with four methods of examining the contents of a variable in a data view window.

1. Press the right mouse button over a variable name in a declaration or executable statement.
2. Move the cursor to a variable name in a declaration or executable statement and press **Ctrl+P**.
3. Mark a block over an expression and either press the right mouse button over the block or press **Ctrl+P** (this allows the expression to be displayed).
4. Use the 'print' command from the command line (this allows complete freedom in the choice of data shown, see page 70).

Methods 1 and 2 provide a very quick way to access simple variables. Methods 3 and 4 can be used to access more complex information like the current array element in a loop. Method 3 has the disadvantage that the expression must be present in the source code. Using the command line (method 4) allows greater flexibility.

To mark a block you can either drag the mouse pointer over the text whilst keeping the left hand mouse button depressed. Alternatively, you can use the arrow keys with the shift key held down. The block is shown as blue text on a cyan background. Pressing an arrow key without holding down the shift key will cancel the block mark.

In addition to the above methods for examining variables, the Win16 and Win32 debuggers provide 'tooltips' which appear when the mouse cursor passes over a variable in the source window. This tooltip takes the form of a small volatile window containing the value of the given variable.

Profiling information

You can display *profiling* information (i.e. information on how many times each line has been executed) for a source file by pressing the F9 key. You must have compiled the source file with `/PROFILE`. The numbers displayed down the right hand column give the number of times each line has been executed. You can also display profile information with the 'profile' command (see page 70).

Only one source file can be compiled with /PROFILE. In addition the profile counts can be written to a file.

Miscellaneous information

The following table summarises the miscellaneous actions available with SDBG

Key	Meaning
F10	Displays assembler output for this source code
F1	Help
Ctrl+F1	Context help. SDBG will examine the text under the cursor and look in the help index. If the word exists that topic is displayed
Right mouse button or Insert	Displays context menu-alternatives to some keystrokes

Variables window

The variables window displays a variables list for the current source window (i.e. the one nearest the top). If the source window is not in the call stack the variables window will be empty. You can switch back to the source window by pressing F4. The window contains a highlight bar that shows the currently selected variable. The type of the variable is displayed in the bottom left corner of the window border.

In addition to using the scroll bar, you can move the highlight bar by using the following keystrokes:

Key	Meaning
Up	Bar up one
Down	Bar down one
Page Up	Bar up one page
Page Down	Bar down one page
Home	Start of list
End	End of list
Right	Scroll the window to the right
Left	Scroll the window to the left

The variables window displays the contents of all the variables in the current scope. This is usually more than adequate for simple variables. It is often useful to have commonly accessed variables or more complex variables (such as C structures or Fortran 90 types) displayed in a separate *data view* window (see below). From a variables window this can be achieved by one of two methods:

1. Press **Enter** with the variable highlighted.
2. Double-click the left mouse button over the variable name.

Data viewing windows

The variables window is always available and allows you to quickly see the current state of variables in a routine. A *data view* window is a window dedicated to one particular variable (or part of a variable) allowing you to see its contents in isolation. There are four different types of data view: *simple expression*, *array*, *structure* and *memory dump*. In addition, from one data view window you can also open other. These five types of data view are described below. You do not have to worry about which view should be displayed. It is all handled by SDBG.

The method used to display a data view window depends upon the current window. Details are given on page 61 for a source code window, immediately above for a variables window, and on page 68 for an existing data view window.

Simple expression

A *simple expression* window is displayed in one of two situations:

1. when the result of the variable or expression is a simple data type that can be displayed in one line, these data types include: integer, logical, real, complex, string and pointers to pointers;
2. when the variable or expression is in error, in which case an error message will be displayed.

If the data type is a pointer then you can display another window (that is the result of dereferencing the pointer) by pressing the **Enter** key. If the data is too long to fit into the window you can scroll the window to the left or right by pressing the left and right arrow keys. If you press the right mouse button with the mouse cursor over an expression window then a menu will appear.

This menu contains the following items:

Menu item	Action
Print value	Same as pressing Enter .
Memory dump at variable	Opens a memory dump window located at the address of the result. For example if the window displayed the value of a variable called <i>ptr</i> , this would produce a memory dump showing the physical memory used to store <i>ptr</i> .
Memory dump using contents	Opens a memory dump window located at the address pointed to by the result of this expression. The result does not have to be a pointer for this to work.
Set write break on variable	Places a write data break on the variable (see page 71).
Set use break on variable	Places a use (read or write) break on the variable.

You can close any data view by pressing the **ESC** key.

Array

An *array view* window will be displayed if the variable or expression results in an array. The array elements are displayed in a column. The window contains a highlight bar that shows the currently selected element. This can be expanded into its own data view window by either pressing **Enter** or double-clicking with the left mouse button. You can move the highlight bar by dragging the scroll bar. In addition you can also move the highlight bar by using the following key presses:

Key	Action
Up	Bar up one
Down	Bar down one
Page Up	Bar up one page
Page Down	Bar down one page
Home	Start of array
End	End of array

If you press the right mouse button with the mouse cursor over an expression window you will see a menu appear.

The items on the menu are as follows:

Menu item	Action
Print value	Same as pressing Enter .
Set visible range	This opens up a dialog that allows the visible range of subscripts to be set. This means that you need only display the array section that you are interested in.
Memory dump at variable	Opens a memory dump window located at the address of the result. In this case this would be a memory dump showing the physical portion of memory used by this array element (and those around it).
Memory dump using contents	Opens a memory dump window located at the address pointed to by the result of this expression. The result does not have to be a pointer for this to work.
Set write break on variable	Places a write data break on the variable (see page 71).
Set use break on variable	Places a use (read or write) break on the variable.

You can close any data view by pressing the **Esc** key.

Structure

A *structure view* window lists the elements of a type (Fortran 90/95), structure (C), union (C), or class (C++) and their values. Each element resides on its own line in a manner similar to the array view and variables list.

The window contains a highlight bar that shows the currently selected element. This can be expanded into its own data view window by either pressing **Enter** or double-clicking with the left mouse button.

In addition to moving the highlight bar by dragging the scroll bar you can use the following key presses:

Key	Action
Up	Bar up one
Down	Bar down one
Page Up	Bar up one page
Page Down	Bar down one page
Home	Start of structure
End	End of structure

If you press the right mouse button with the mouse cursor over an expression window a menu will appear.

The items on the menu are as follows:

Menu item	Action
Print value	Same as pressing Enter .
Memory dump at variable	Opens a memory dump variable located at the address of the result. In this case showing the memory taken by this structure member.
Memory dump using contents	Opens a memory dump window located at the address pointed to by the result of this expression. The result does not have to be a pointer for this to work.
Set write break on variable	Places a write data break on the variable (see page 71).
Set use break on variable	Places a use (read or write) break on the variable.

You can close any data view by pressing the **ESC** key.

Memory dump

A *memory dump* window shows the individual bytes of memory with no formatting. The data in this window is displayed in three columns. The first column contains the start address of a strip of memory. The second column shows the bytes of memory that are contained in the memory starting at that address. The third and final column contains the ASCII representation of the same strip of memory.

The width of the strip depends on the window size and will automatically be scaled to the size of the window. In addition you can quickly change the size by pressing one of the following keys:

Key	Width
6	16 bytes wide
8	8 bytes wide
4	4 bytes wide
2	2 bytes wide
1	1 byte wide

Data values that read 'XX' constitute an invalid address.

One of the data values is highlighted. This is the current address. This is initially set to the address that you requested to display. The highlighted address is mirrored in the ASCII representation.

You can move the highlight using the following keys:

Key	Action
Left	One byte to the left
Right	One byte to the right
Up	Up one line
Down	Down one line
Page Up	Up one window height
Page Down	Down one window height

Pressing **Alt+P** will take the byte under the highlight and the following three bytes to form an address. The window will then be refreshed using this new address. This allows a pointer to be followed. You can go back to the address at which you pressed **Alt+P** by pressing **Alt+B**. You can nest **Alt+P** key presses to a depth of 20 and still be able to return to the starting point using **Alt+B**.

The expression, structure and array view windows all update to show any new values whenever the program is stepped or run to a new point. Because you may be looking at a specific area of memory, the memory dump window does not do this automatically even if the value of the expression used to set the window changes. You can force the window to reposition itself in memory by pressing **Ctrl+O**.

If you press the right mouse button with the mouse cursor over a memory dump window you will see a menu appear.

The items on the menu are as follows:

Menu item	Action
16 bytes per line	16 bytes per memory strip
8 bytes per line	8 bytes per memory strip
4 bytes per line	4 bytes per memory strip
2 bytes per line	2 bytes per memory strip
1 byte per line	1 byte per memory strip
Set write break	Places a write data break on the address
Set use break	Places a use (read or write) break on the address

Data view window

It is possible to open a data view window from a previous data view. This is used when following pointers etc.. To open one data view from another:

1. press **Enter** with the item highlighted or
2. double-click the left mouse button over the item name.

You can close any data view by pressing the **Esc** key.

Machine code windows

A *machine code* window displays the instructions that the CPU uses to execute your program and should only be used by people who understand assembler. A machine code window will be displayed on the following occasions:

- if you select a routine in the call stack that has no debugging information; these routines have grey lettering rather than black,
- if you select a routine in a *Find* window that has no debugging information; these routines have the words '(no debugger information)' following the routine name,
- if you press **F10** from a source window (**F11** for the Win16 and Win32 debuggers).

As with source windows the current execution point is shown by a red bar. An execution point that is not at the top of the call stack is shown in brown.

The window is split into three distinct columns. The first column shows the start address that the instruction is located at. The second column shows the assembler instruction at that location and the third column shows the offset of the instruction into the routine. The following key presses can be used within this window:

Key	Action
Up	Move up one instruction.
Down	Move down one instruction.
Page Up	Move up one page of instructions.
Page Down	Move up one page of instructions.
Ctrl+Home	Move to first instruction in routine.
F7	Step one instruction.
F3	Get to the instruction the cursor is indicating.
F2	Set a machine level breakpoint at the instruction the cursor is indicating.

F10	Display source code if debugging information is available.
Alt+R	Display registers window.

Command line

To provide greater flexibility within SDBG a simple command line facility is available. The command line is accessed from the source window. It can be displayed by pressing **Alt+P** in the source window. Alternatively, you can display the command line by pressing any alphanumeric key (without holding down **Alt** or **Ctrl**). In this case the key press will appear on the command line. You can hide the command line by pressing **Alt+P** again. It is also possible to edit the command line.

The following key presses are permitted when editing a command line:

Key	Action
Left	Cursor left
Right	Cursor right
Home	Start of line
End	End of line
Backspace	Delete character to left
Delete	Delete character under cursor
Up	Recalls last command line (up to 20 are stored)
Down	Recalls next command line (up to 20 are stored)
Enter	Execute command line
Esc	Clear command line. If the command line is already clear it will be hidden.

Commands

This section contains a list of the valid commands that may be entered on the command line.

L/*text*/ or **/***text*

This command performs a search forward within the source window for *text* and repositions the cursor if the text is found. The search is case insensitive.

BL*/text/* or **?text**

This command performs a search backward within the source window for *text* and repositions the cursor if the text is found. The search is case insensitive.

MOVETO *n*

Moves the cursor to line *n*. If *n* is greater than the number of lines within the file then the command is ignored.

PROFILE

Toggles profile information and has exactly the same effect as pressing F9.

PROFILE *filename*

This will write the source window, together with any profile counts, into the file *filename*. The profile information must already be displayed.

PRINT *expr* or **P** *expr*

Produce a data view window for the expression *expr*. The type of data view displayed is dependent on the expression given and will be automatically adjusted. This command provides a more general mechanism than displaying individual variables or marked expressions.

Examples

```
print machine_data[i]    - C syntax
```

```
print machine_data(i)    - Fortran syntax
```

```
print mph*1.6
```

```
print *ptr+stru->element - C syntax
```

PRINTMEM *expr* or **PM** *expr*

Produce a memory dump window centred about the value of *expr*. The given expression does not have to be a pointer type. It can be an integer or even a calculation.

VIEW *filename*

This command opens a new source window and displays the file *filename* in it. If the file is not an object file that makes up the current program or the relevant object file does not have debugging information then you will not be able to display expressions or set breakpoints from it. Any ASCII file can be displayed with this command.

FIND *routine*

This command will search for routines whose names contain the text *routine*. If one match is found that routine will be displayed. If more than one routine name matches the text given then a *Find* window will appear displaying all the matches.

The window will respond to the following key presses:

Key	Action
Up	Move the highlight bar up one line
Down	Move the highlight bar down one line
Home	Move the highlight bar to the top of the list
End	Move the highlight bar to the end of the list
Page Up	Move the highlight bar up one page
Page Up	Move the highlight bar down one page
Enter	Display the routine. If the words ‘(no debugging information)’ appear after the routine name then a machine code window will be displayed. Otherwise the file appearing in brackets after the routine name will be displayed.
Esc	Close the window

The *Find* window will be kept open to allow further selections to be made, although it will probably be initially hidden by the new source (or machine code) window. You can easily cycle through all open windows by pressing **Alt+N**.

Example

```
find init
```

Could find the routines `initialise_module`, `AnswerInIteration` and `D0INIT`

WRITE_BREAK *expr* or WB *expr*

Places a write data break on the address indicated by evaluating *expr*. The break point will be set on the address of the result. For example:

```
write_break count      breakpoint placed on address of count
write_break *ptr       breakpoint placed on value of pointer ptr
write_break arr(9)     breakpoint placed on ninth element of arr
write_break 0x78647292 breakpoint placed on address 0x78647292
```

USE_BREAK *expr* or UB *expr*

Places a use (i.e. read or write) data break on the address indicated by evaluating *expr*. The break point will be set on the address of the result as is the case with **WRITE_BREAK**.

REGS

Displays a window that shows the current values of the CPU registers. The values are in hexadecimal. The floating point stack is also shown.

BREAKPOINTS or BPS

Displays a window which contains the status of currently active breakpoints.

STREAMS

Opens a window that lists the currently open Fortran units.

STREAM *n*

Opens a window showing the status of Fortran unit *n*.

LET *expr1=expr2*

This command allows you to make changes to data without having to recompile. The value of *expr2* is assigned to the item indicated by *expr1*. *expr1* must be an expression to which a value can be assigned, for example `let 12=6+a` is invalid. If the two expressions refer to different data types a conversion will be applied to the result of *expr2* to allow it to be used. You should, however, exercise caution when using differing types.

Examples

<code>let i=10</code>	Simple variable assignment
<code>let arr(j)=arr(count)</code>	Array element assignment
<code>let shape.colour=0xf3229</code>	Structures

DOS *cmdline* or EXECUTE *cmdline* or X *cmdline* (DOS debugger only, not Win16/Win32)

This will load the command processor and execute the command *cmdline*, which may be a standard '.COM', '.EXE' or '.BAT' file. The command line may be omitted in which case the command shell will be started into which you can type commands. You should type the command EXIT to return to SDBG. You should not execute commands which:

- ☐ Modify (or attempt to modify) any open files. This includes removing disks from the floppy drive that your program is using.
- ☐ Try to execute Microsoft Windows or DosShell.
- ☐ Execute any TSR program, including network shells.
- ☐ Run any DBOS application. This includes Salford compilers and linkers.

Customising the debugger

The debugger can be customised in order to change its look and feel. Under DOS the keystroke **Alt+O** will display the options. The Win16/Win32 debuggers use an Options entry in the Tools menu. The options available are:

Automatically open variables window – When checked the *variables* window will open automatically when the debugger is entered. When debugging programs with large numbers of variables it can sometimes be better to not open the *variables* window and just use the *data view* windows or tooltips. Default is on.

Sort variables window alphabetically – When checked the *variables* window is sorted into alphabetical order. When not checked the variables are listed in scope order, that is, local variables will be listed first followed by globals. Default is off.

Show PARAMETERS in variables window – When checked PARAMETERS will be shown in the *variables* window. When debugging programs with large numbers of parameters (Windows applications in particular) the parameters can clutter up the *variables* window, obscuring the variables. Default is on.

Only use one source window – With this option turned on the debugger will only use one window for displaying source files. When turned off the debugger will use a new window for each routine in the call stack that is shown. Default is off.

Show Tips at Startup – When checked the debugger will show the ‘Tip of the day’ window at startup. Default is on.

Debugger is MDI – When checked the debugger windows will be enclosed within a MDI (multi document interface) window. When not checked the windows will appear directly on the desktop. Win16/Win32 only, default is on.

Display bubble help – When checked the toolbar buttons will show tooltips when required. Win16/Win32 only, default is on.

Display variable values in source – When checked popup tooltip help will appear when the mouse cursor is over a variable name. The tooltip help will contain the variable’s name and value. Win16/Win32 only, default is on.

Program development

Diagnostic facilities

FTN77 provides extensive diagnostic facilities which enable programs to be speedily developed and debugged. Diagnostics can be output

- ☐ during compilation,
- ☐ during loading,
- ☐ at run-time.

These three types of diagnostics are described separately below.

Compilation diagnostics

During compilation, three types of messages can be output:

- 1) **ERROR MESSAGES** which indicate that the rules of Fortran 77 have not been obeyed, for example, that a label has been referenced but not defined. Error messages are preceded by *** (three asterisks).

It is possible (but not recommended) to load and execute a program that contains compilation errors (if the **/PERSIST** option was used) but unpredictable results will occur if the parts that are executed contain compilation errors. If **/PERSIST** is not used, the compiler will cease code generation once an error has been reported and the relocatable binary file will be marked to make it unloadable.

Note that certain error conditions become fatal when the **/ANSI** option is used otherwise they are classed as warnings.

2) **WARNINGS** are output for one of two reasons:

- If the program is correct Fortran 77 but probably contains a logic error. For example, the following statement is legal but will cause an infinite loop:

```
10 GOTO 10
```

In the following example, the compiler will warn that the second statement will never be executed.

```
RETURN
A = B
C = D
. . .
```

- Each time the program uses those extensions to Fortran 77 (see chapter 13) which have been included in order to allow compatibility with Fortran 66.

For example, users converting programs containing Hollerith data will find their listings annotated with the message:

```
Warning: The use of Hollerith data is an extension to
Fortran 77.
```

It is always possible to load and execute a program whose compilation produces only warnings.

3) **COMMENTS** are informative messages. They serve to remind the programmer that there might be a better way of writing a particular statement. As an example, the statement

```
A = FLOAT(I)
```

would cause the compiler to output the message:

```
COMMENT: FLOAT could be replaced by its generic
equivalent (REAL) throughout this program unit
```

Most messages are output immediately after the statement to which they refer.

If it is necessary to delay the output of a message or the source listing option (see page 23) has not been chosen, the message is followed by a line number which refers to the source file. Certain error messages referring to **EQUIVALENCE** statements are always output (with a line number reference) immediately after the first executable statement in a program unit has been listed.

Some messages, notably those referring to undefined or unused labels, are not output until the **END** statement of a program unit has been processed.

Each diagnostic message has an associated error number. It is possible to instruct the compiler to ignore every occurrence of the error associated with a particular error number by using the **//IGNORE** compiler option as follows:

```
FTN77 MYFILE /IGNORE <error number>
```

where <error number> is the number of the error that is to be ignored. This number can be obtained by using the /ERROR_NUMBERS compiler option in an earlier compilation that exhibits the error. More than one /IGNORE option can be specified, if it is desired, in order to ignore several errors.

Note:

If messages other than warnings or comments are ignored, the compiler may generate incorrect code.

An example of a source listing containing errors, warnings and comments appears in Figure 8-1.

```
SALFORD UNIVERSITY FTN77-VER. x.xx  C:\PROJECT\MYPROG.FOR

COMPILER OPTIONS: LISTING INTS NOMAP NOCHECK LOGS DYNM OFFSET
NOANSI PAGETHROW NOSILENT NO_OPTIMISE

0001          OPTIONS(NOCHECK)
0002          GOTO 1
0003          DO 10 I=1,6                                AT 0022
WARNING - This statement will never be executed
0004.01          J = 3                                    AT 0037
0005.01          I = J + 1                                AT 003D
*** I is currently in use as a DO or implied DO variable
0006.01  4          K = 6                                    AT 003D
0007.01 10          CONTINUE                                AT 003D
0008          IF(I.EQ.3)THEN                                AT 003F
0009.01          P = Q + R                                    AT 003F
0010.01  1          L = 7                                    AT 003F
0011.01          ENDIF                                      AT 003F
*** Label 1 has been referenced from outside the DO-loop,
    IF, ELSEIF or ELSE block in which it appears
0012          IF(I.EQ.4)THEN                                AT 003F
0013.01          DO 20 K=3,7                                AT 003F
0014.02          A = FLOAT(I)/B                              AT 003F
0015.02          END                                        AT 003F
*** Unterminated DO statement(see line 13)
*** Unterminated block-IF statement(see line 12)
*** Label 20 has not been defined
```

Figure 8-1 Error and warning messages

Linker diagnostics

During the loading of a program, the relocatable binary code that has been output by the compiler is linked with routines from the Fortran 77 library and from other relocatable binary files and libraries specified by the user. There are a number of error and warning messages that can be output by the linker, most of which are self-explanatory.

A commonly occurring message is one that reports that a routine is missing. A name can appear as “missing” for either of the following reasons:

- 1) A routine of the specified name is not available to the loader because:
 - an appropriate **LIBRARY** directive (see page 41) has not appeared in the source program or
 - the name of a library routine has been misspelt. A commonly occurring error is the use of the letter O instead of the digit 0 in calls to library routines, for example, the use of MO1ANF instead of M01ANF.
- 2) The name was intended to be an array element name but has not been dimensioned. It has then been used only in a function reference, a **CALL** statement or on the right hand side of an assignment statement, for example:

```
B = A(3)
CALL SUB(A(I),X)
C = F(A(I+J))
```

Fortran is defined in such a way that each of the above would generate a reference to a function called A. The name A would be output by the loader as “missing”.

Note:

If the “missing” name corresponded to a routine in a library compiled in **CHECK** mode, a run-time error might occur saying that the routine had been called inconsistently. In the worst case, an appropriate routine with consistent arguments would be loaded and the program would run with unpredictable results!

Programs with missing routines *can be executed* up to the point at which a missing routine is called.

Run-time diagnostics

Comprehensive run-time diagnostic facilities are provided by the system in such a way that users can always choose the level of checks that are applied to any part of their program.

During the early stages of program development, it is useful to have all or most of these checks performed by the system but later, when the program is thought to be thoroughly tested, it is usual to remove checks in order to achieve the fastest possible execution speed and smallest possible object program size. If new routines or lines of code are added to an existing program, it is a simple matter to specify that checks should be performed only on the program units that have been changed.

The available run-time diagnostic information is controlled by directives which may appear before any program unit. Note that the default level of checks to be applied can be set by one of the FTN77 compile-time options `/CHECK`, `/FULLCHECK`, `/UNDEF` or `/NO_CHECK`. These keywords may also appear as part of an `OPTIONS` directive.

For example:

```
OPTIONS (CHECK)
OPTIONS (UNDEF, FULLCHECK)
```

Once an error has been detected by the checking mechanism, execution terminates and the system enters the symbolic debugger to give diagnostic information.

The run-time checks are described more fully in the sections which follow.

Arithmetic overflow checking

No computer permits the storage and manipulation of arbitrarily large quantities. The following limits apply when using FTN77:

INTEGER*1	-128 to +127
INTEGER*2	-32768 to +32767
INTEGER*4	-2147483648 to +2147483647
REAL (REAL*4)	$\pm(1\text{E}-37 \text{ to } 1\text{e}+39)$ (approx.)
DOUBLE PRECISION (REAL*8)	$\pm(1\text{D}-307 \text{ to } 1\text{D}+309)$ (approx.)

If a calculation is performed whose result exceeds these limits *arithmetic overflow* occurs.

If a `CHECK` or `FULLCHECK` directive appears in the source program, then runtime checking for overflow is enabled.

If a checked statement does set overflow then execution is terminated and the interactive debugger is entered (see chapter 7). If a statement sets overflow and is not checked, then execution continues with an incorrect result in the case of integers, but terminates in the floating point case.

When a program is loaded, all numeric variables (except those which have appeared in a **DATA** statement) are initialised to an “undefined” value unless the **/ZEROISE** compile-time option is used (see chapter 6).

In the case of integer variables, the undefined value chosen is -32640 which will not result in overflow being set as the result of an assignment and, furthermore, overflow will not always occur when an expression is evaluated which involves an undefined value.

Undefined variables can be trapped by use of the **/UNDEF** option (see below).

Note:

Variables and array elements in otherwise uninitialised common blocks are not initialised to the undefined value.

Argument consistency checking

There are a number of run-time checks associated with the calling of routines. A subroutine or function compiled with a checking option will produce a run-time error if one of the following occurs:

- 1) Arrays used as actual arguments are too small for the declared size.
- 2) An actual argument which is a constant or a local variable that is in use as a **DO**-variable is altered by the called routine. For example:

```
      CALL FRED(1.0)
      . . .
DO 10 I=1,100
      CALL FRED(I)
      . . .
10  CONTINUE
      . . .
      END
      SUBROUTINE FRED(N)
      . . .
      N = G
      . . .
      END
```

Either of the calls to **FRED** in the above example would cause a run-time error.

- 3) A simple character argument is not large enough for its declared size.

In the absence of checking these conditions result in program corruption with unpredictable results.

Array subscript checking

The `/CHECK` option ensures that every array reference lies within the storage allocated to the array. Each individual subscript expression is only checked if `/FULLCHECK` is specified. Consider the following coding:

```
DIMENSION A(10,10)
I = 11
J = 7
A(I,J) = 0.0
```

The storage element referenced by the subscripts lies within the declared storage for the array even though the first subscript is outside its corresponding bound. This is not valid Fortran 77 (although it is valid Fortran 66). In this example, a run-time error would only be produced by the use of `/FULLCHECK`. `/CHECK` would not produce a run-time error.

Using the above `DIMENSION` statement for `A`, it is apparent that the statement

```
I = 11
J = 10
A(I,J) = 0.0
```

would cause a run-time error if either of the compiler options were used.

In general, array bound checking incurs a run-time overhead of both store and execution speed. Full array bound checking for multi-dimensional arrays is very costly. The simpler array bound check is less so.

Array bound checking is available for arrays of any type. The array may have explicit dimensions, for example:

```
PARAMETER (N=10,M=6)
DIMENSION A(N,M),B(10,20)
```

or may be passed as arguments with variable bounds, for example:

```
SUBROUTINE FRED (A,B,C,N)
COMMON/ABC/M
DIMENSION A(M),B(N),C(*)
```

The checks will work in all cases for both upper and lower bounds.

If checking is not in use, unpredictable effects may occur at run-time. An attempt to transfer a value from an element outside the bounds of an array can either:

- 1) assign or use an arbitrary value which might cause overflow, or
- 2) cause the program to fail with general protection fault which means that the program has tried to access storage outside the limits available to it, or
- 3) overwrite a pointer and cause a fault in a different part of the program.

If an attempt is made to transfer data to an element outside the defined bounds of an array without specifying the checks, the effects are totally unpredictable and will frequently result in a spurious error when some unrelated part of the program is executed.

The `/OLDARRAYS` compiler option allows Fortran 66 programs that contain constructs such as

```
SUBROUTINE FRED(A,B,N)
  DIMENSION A(1),B(N,1)
  . . .
```

to be treated as

```
SUBROUTINE FRED(A,B,N)
  DIMENSION A(*),B(N,*)
  . . .
```

Use of this option allows array subscript checking to work according to the size of the actual argument array.

Checking for undefined variables (/UNDEF)

`/UNDEF` (which implies `/CHECK`) causes FTN77 to plant code to check that a variable or array element used in the circumstances described below has been previously given a value.

`/UNDEF` causes extra code to be planted for a name or array element appearing in the following circumstances:

- ☐ as the right hand side of a non-character assignment,
- ☐ in arithmetic expressions involving `+` `-` `*` or `**`,
- ☐ in relational expressions involving `.NE.` `.EQ.` etc.,
- ☐ in logical expressions involving `.AND.` `.OR.` etc.,
- ☐ as an array subscript,
- ☐ as a substring expression,
- ☐ as the argument to an ANSI standard intrinsic function such as `SIN`, `COS` etc.,
- ☐ as the expression used within a logical or arithmetic `IF` statement.

`/UNDEF` currently has no effect on character assignments or concatenations.

All local static variables are predefined to an undefined value. This value has HEX 80 in every byte. Routines compiled with `/CHECK` also clear their dynamic variables to this value on entry to the routine. This value is treated as undefined by the

symbolic debugger, see page 49. An undefined integer has one of the following values:

INTEGER*1	-128
INTEGER*2	-32640
INTEGER*4	Z'80808080'

In rare cases, most likely when using integer data, the undefined integer value may be intended by the programmer and the use of `/UNDEF` will cause a spurious error to be reported. In this case, all that can be done is to compile the program unit(s) in question without `/UNDEF`. Note that:

- The use of `/UNDEF` causes a significant run-time execution speed penalty.
- It is necessary to compile the *main* program with this option if uninitialised common blocks are to be set appropriately.

ASSIGNED GOTO statement checks

FTN77 ensures that if a local variable is used in an assigned `GOTO` statement there is at least one `ASSIGN` statement for the variable in the program unit. Thus, for example

```
CHECK
J = 3
. . .
GOTO J
```

would cause a compile-time error. Run-time checks are provided to ensure that:

- 1) if a label list is present, the integer variable is currently `ASSIGNED` to a label in the list and
- 2) if no label list has been specified, the transfer of control is within the current program unit.

Note that this check is not watertight and that a program which attempts to `GOTO` an integer whose value happens to lie within the range of the routine will go out of control even in `CHECK` mode.

Character data

The checking mechanism provides the following diagnostic checks for character data:

- 1) That an argument of type character is of sufficient length for its declared dummy size. For example, in `CHECK` mode, the following program would cause a run-time error:

```
CHARACTER*20 A
. . .
CALL CHSUB(A)
. . .
END
SUBROUTINE CHSUB(X)
CHARACTER*30 X
. . .
END
```

The error could be prevented in this case by declaring *X* in the subroutine as follows:

```
CHARACTER*(*) X
```

so that *X* would assume the character length of the actual argument.

- 2) That substring expressions are valid. There are two possible sources of error that may arise when using a substring reference of the form *A(I:J)*:

- ☐ either the value of *I* is greater than the value of *J* or
- ☐ the value of *I* is less than 1 or the value of *J* is greater than the declared or assumed length of the character variable or array element.

All of the character assignment statements in the following program would cause a run-time error:

```
CHARACTER*20 A,B(20)
. . .
I = 0
J = 21
K = 4
L = 3
A(I:) = 'X'
A(1:J) = 'XXX'
B(3)(K:L) = 'XXX'
END
```

9.

Optimisation and efficient use of Fortran

Introduction

This chapter describes the FTN77 local and global optimisation features and indicates some of the ways in which a programmer can write Fortran programs that will make the best use of these features.

Optimisation

The /OPTIMISE compiler option

/OPTIMISE selects the optimisation facility described below. For those installations where /OPTIMISE is the chosen default /NO_OPTIMISE is provided.

The alternative spellings, /OPTIMIZE and /NO_OPTIMIZE, are provided for those who use a well-known alternative version of the English language!

The /OPTIMISE option causes the compiler to make a second pass through the source code image in order to perform improvements to the object code that will result in faster execution times for typical programs.

Under DOS, /OPTIMISE may be used together with the /WEITEK compiler option.

Using a coprocessor

The compiler will automatically generate correct code for an Intel compatible numeric coprocessor. Under DOS, the compiler will also generate code for a WEITEK

numeric coprocessor when the `/WEITEK` compiler command line option is used. Support for the 80287 coprocessor has been discontinued.

DBOS will support both an Intel compatible coprocessor and the WEITEK coprocessor. However, if you wish to enable DBOS's ability to use the WEITEK coprocessor you must add the `/WEITEK` DBOS command line option. No version of Windows supports WEITEK numeric coprocessors.

Optimisation processes

The improvements in execution speed that are obtained depend upon the style and content of the source program, for example, whether one- or multi-dimensioned arrays are used, whether nested loops appear, and so on.

As optimisation can involve source code re-arrangement and a change in the way that registers and store locations are used, it is possible that numerical results produced by an optimised program may differ in some way from those produced by the unoptimised version of the same program. This effect may be more noticeable with iterative algorithms and is due to the fact that a more accurate value can be held in a coprocessor floating point register than can be held in the corresponding store location.

Some programs may actually execute more slowly when optimised due to non-executed loops that cannot be detected by the compiler, for example:

```
DO 10 I=1,N
```

where **N** is zero or negative at run time. In this case code that is moved out of the loop will be executed once, rather than not at all as would happen if this optimisation had not been made.

When the compiler option `/OPTIMISE` is used, the compiler performs code optimisation based on rearranging the order of execution of statements which constitute a program unit (see below). If `/OPTIMISE` is not used, the following optimisations are typical of those performed by default.

- Constant 'folding' and conversion of Fortran type at compile time. Constant folding is the process of taking a statement such as:

$$A = I + 3 + 7$$

and producing code which is the same as for the statement:

$$A = I + 10$$

This might not appear to be of much use at first glance, since you might not think that you would write expressions with multiple constants in this way. However, consider the expression $2*PI*R$ where **PI** is a parameter - the $2*PI$ part would be evaluated at compile time. In addition to this however, a number of situations arise for the implicit arithmetic which the compiler plants code for (chiefly array

subscript calculation) where this technique results in considerable reduction in the amount of arithmetic done at run time.

Related to this is the conversion of the type of constants where appropriate. For example, the statement:

$$X = 4$$

is compiled as:

$$X = 4.0$$

thus the need for a type conversion at run time is obviated.

- Elimination of common subexpressions within a statement. Again, this applies equally to expressions which form subscript calculations. Consider the following assignment:

$$A(I, J+K) = A(I, J+K) + 3$$

The code necessary to calculate the offset represented by (I,J+K) is only performed once.

- The contents of registers are “remembered” between statements so that redundant load and store operations are avoided. For example, consider the following sequence of statements:

$$\begin{aligned} K &= I + J \\ L &= K * I \end{aligned}$$

For the second statement, the compiler recognises that it has the value of K in a register, so it does not need to load K from store.

Note, however, that it will probably need to reference the value of I from memory, since the calculation of I + J will have resulted in the loss of the value of I from a register.

Even if there were some statements interspersed between the statements above, this optimisation could still take place, so long as:

- the register in question was not used for another purpose in the interim, and
- none of the interim statements were GOTOs, and
- none of the executable statements were labelled (a good reason to dispense with unused labels in your code).

The compiler tries to avoid using registers which might contain something useful in a subsequent calculation.

A related technique is used for the coprocessor floating point registers, although due to the limited size of the hardware register stack, it is not possible to leave a value in a register just in case it might be useful. Instead, if a recently calculated

floating point value proves to be useful for a subsequent calculation, the instruction which places the result in the corresponding memory location is converted from “store and pop” to “store and don’t pop”. The value is then available somewhere in the register stack for the subsequent calculation.

Note that this floating point register tracking is not performed when the `/DEBUG` compiler option is used or implied. Furthermore, the compiler option `/NO_FLOATING_TRACKING` can be used to disable this process (see page 94).

- Full use is made of the instruction set. For example, an integer addition or subtraction of 1 is performed by the appropriate increment or decrement instruction. Also, some optimisations can be used to perform certain arithmetic operations a little quicker. For example, evaluation of $I*5$, where I is of integer type, can be performed with the instruction sequence:

```
MOV    EAX%, I
LEA    EAX%, [EAX%+EAX%*4]
```

which is faster than the corresponding integer multiply. Note however, that this optimisation is not done in `CHECK` mode, since any overflow would go undetected.

When the `/OPTIMISE` option is used, optimisations performed include the following:

- 1) Loop invariant motion. This means that any calculations which are constant with respect to the loop variable may be moved, so that they are performed once and for all on entry to the loop. This leads to the actual degradation in performance mentioned earlier, for the case where the loop is not executed at all. However, in most cases, particularly when the loop is executed a large number of times, considerable savings can result.
- 2) Loop induction weakening. This means that, instead of using multiples of the loop index, a constant is added to a pseudo variable each time round the loop. For example, consider the following loop:

```
DO I = 1, N
  A(1, I) = 0
END DO
```

The offset into the array `A` will be a constant multiple of the loop variable `I`. The constant is related to the size of the first dimension of the array `A`. Induction weakening will replace multiplication by this constant to produce the array offset at each iteration of the loop by a faster addition of the constant at each iteration.

- 3) Elimination of common subexpressions across statements. This is often a consequence of the optimisations in (1) and (2) : expressions which are taken out of the loop as either loop invariant, or as candidates for induction weakening, can themselves be sub-parts of larger expressions.

- 4) In some loops, particularly useful quantities can be “locked” into registers. “Locking” means that, for the duration of a loop, the value of a program variable, or perhaps a derived quantity such as an offset into an array, is kept in a register, and is not stored into its associated store location (if indeed it has one) until exit from the loop.

Obviously, this requires that exit from the loop cannot be by means of a **GOTO** from within itself, and that no subroutine or function is called from within the loop, as these statements could destroy any value held in the register.

Also, there is some trade-off involved in tying up a register in this way, so generally locking will only occur for relatively short loops.

Optimisation of the loop in the example given in 2) above involves induction weakening and locking the array offset in a register.

- 5) Some additional optimisations based on the 80486 and Pentium instruction set. In some cases integer instructions are used instead of floating point instructions. This often results in different behaviour where the operands are invalid (for example where they should cause an overflow), but it is assumed that, if optimisation is being employed, problems such as this have been eliminated.
- 6) Many cases of a “dot-product” construction are spotted and replaced with faster code, for example:

```
DO I = 1, N
  SUM = SUM + A(I)*B(I)
END DO
```

This is particularly efficient when optimisation is used in conjunction with the **/WEITEK** option - the Weitek “multiply and accumulate” instruction is used.

- 7) Many cases of redundant combinations of instructions are eliminated, for example, jumps to the next line, loads from a register to itself which sometimes are generated as a result of register locking (see 4 above).

The above list is not exhaustive, and new optimisations will be added during the course of compiler development.

It is possible to further improve the execution speed of certain programs by using the **/UNSAFE** option in conjunction with **/OPTIMISE**. **/UNSAFE** allows the compiler to assume that the following classes of variable and array names can be subjected to code re-arrangement techniques:

- ☐ Equivalenced variables and arrays.
- ☐ **COMMON** variables and arrays.
- ☐ Argument variables and arrays.

With each of these categories of objects, there may be more than one way to access the storage represented by the object. Thus, in the most general case, it is necessary to

assume that the register from which such a value was recently saved is not necessarily valid, since the storage in question might have been changed via another route. Similarly, any quantity calculated from an object in these classes may not be induction weakened, locked into a register, and so on. The `/UNSAFE` option allows the compiler to assume that such objects will not be changed by their alternate routes (and thus requires some care from the programmer).

Helping the optimiser

The success which the optimiser has with your code depends to a large extent on the code itself. In order to ensure that the object code is correct in all cases, the optimiser takes a conservative approach which can sometimes mean that potential optimisations are ignored. As a rule of thumb, the more structured the code appears to the optimiser, the more optimisations it can apply. It is difficult to give hard and fast rules as to how best to maximise the optimisation which can take place, but a number of general points should be noted:

- GOTOs can often inhibit optimisation. This is particularly the case in tight loops. You may be able to achieve the effect you want by using a logical variable.
- Function and subroutine calls within loops prevent many optimisations from occurring. Apart from the fact that no register tracking can take place across `CALL` statements and function references (the called routine does not save the register set), many of the loop optimisations cannot take place.

Even if the `CALL` statement or function reference appears to be “loop invariant” in some sense (for example, all of its arguments are themselves loop invariant), the `CALL` statement or function reference cannot be moved because of side effects which the routine may have, or common variables which it uses which are not loop invariant.

Thus, it is up to you to remove `CALL`s and function references which are genuinely loop invariant from out of your loops.

- It is a good idea to remove all redundant labels (these are automatically indicated by FTN77's compilation diagnostics). See below.

Efficient use of Fortran 77

Labels

The compiler outputs a warning message if a label has been set but never used. Redundant labels should be removed as their presence inhibits optimisation in many cases. Labels can also often be removed by making small changes to the structure of the program, for example:

```

      IF(I.NE.0)GOTO 10
      A = B
      C = D
10    J = I

```

If label 10 had not been referenced from elsewhere in the program unit, this could be rewritten more efficiently (and legibly) using a **block-IF** statement as follows:

```

      IF(I.EQ.0)THEN
        A = B
        C = D
      ENDIF
      J = I

```

The extra efficiency would derive from the fact that the compiler ‘remembers’ that it has I in a register when it compiles the statement J=I.

Labels may also often be removed from a program by removing arithmetic IF statements, especially when two of the labels are the same, for example:

```

      IF(I-J)1,2,2
1     B = 3

```

would be better written:

```

      IF(I.GE.J) GOTO 2
      B = 3

```

Intrinsic functions

The following intrinsic functions are compiled as in-line code:

- Type conversion functions such as INT, REAL, DBLE, CMPLX, CHAR, ICHAR, CONJG and DIMAG.
- BITS, AND, OR, XOR, NOT, LS, RS, LR, RR, SHFT, LT, RT, LOC, CCORE1, CORE1, CORE2, CORE4, FCORE4 and DCORE8.
Note: these functions are FTN77 extensions.
- The MAX and MIN functions.
- ABS and its non-generic variants, except CABS and CDABS.
- LEN and LENG.
- LGE, LGT, LLE and LLT.
- The long to short conversion functions INTB, INTS, INTL, LGCB, LGCS and LGCL.
- Under DOS/Win16, SQRT (unless you have a 386 with a Weitek 1167 coprocessor, then SQRT is *not* performed in-line).

□ Under DOS/Win16, SIN, COS and TAN (unless a Weitek coprocessor is used).

□ INDEX, if the second argument is of length 1, for example:

```
K=INDEX(MESSAGE, ' ')
```

Statement functions

Statement functions are always expanded as in-line code. Efficient execution is therefore guaranteed and is to be preferred to the supplying of a one line external function.

Common subexpressions

In most cases, common subexpressions are evaluated only once. Thus the following code could not be improved by the prior assignment `TEMP=X*Y`:

```
Z = (X*Y)/(1.0+X*Y)
```

Common subexpressions may sometimes be evaluated more than once in character expressions and in arithmetic expressions contained in logical IF statements.

Constants

The constant parts of expressions are evaluated at compile-time so that `PARAMETER` statements can be used in many cases to make programs more readable without increasing execution time. For example, consider the following:

```
PARAMETER (PI=3.14159)
      . . .
      CALL FRED(PI/2.0)
```

The expression `PI/2.0` is constant and is therefore evaluated at compile-time and nothing would be gained by replacing the expression with its calculated value.

Dummy array dimensions

It is more efficient to dimension a dummy array `A(*)` rather than `A(N)` if the value of `N` implies the whole of array `A`.

Character variables

The manipulation of long character data has hidden overheads. In particular, consider the following:

```
CHARACTER*100 A
      . . .
      A = 'FRED'
```

The execution of the assignment statement involves the insertion of 96 blanks to pad out the variable *A* to its declared length of 100. Note, however, that

```
A = ' '
```

is far more efficient than for example:

```
      . . .
      DO 1 I=1,100
1      A(I:I) = ' '
      . . .
```

Format statements

Unlike many Fortran implementations, FTN77 preprocesses ‘constant’ formats at compile-time. These ‘constant’ formats are as follows:

- A **FORMAT** statement.
- A format expression that is a character constant or a character constant expression.
- A format expression that is a parameter name.

All formats which include character arrays, array elements or variables are decoded at run-time. Such non-constant formats require more extensive decoding which leads to longer execution times.

For example, the following should be avoided wherever possible:

```
CHARACTER*10 F
F = '(3F10.4) '
      . . .
WRITE (2,F)X,Y,Z
```

It could be rewritten as follows:

```
CHARACTER*10 F
PARAMETER (F='(3F10.4) ')
      . . .
WRITE (2,F)X,Y,Z
```

so that the format specifier would be decoded at compile-time.

Note also that the colon (:) edit descriptor and tab facilities can often be used instead of a run-time format.

Switching off variable tracking

On page 85 it was noted that by default (even when the compiler option `/OPTIMISE` is not used), the use of coprocessor floating point registers is tracked in order to eliminate unnecessary register reloading instructions. However, in certain very unusual circumstances, the use of variable tracking could have undesirable side effects. For this reason, the compiler option `/NO_FLOATING_TRACKING` has been made available to turn off register tracking for floating point values.

For example in extreme circumstances a problem may arise because a register value is stored with greater precision. The following code illustrates this feature.

```
      X=1.0
      Y=X+1E-8
C      Normally you would not expect to be able to hold nine
C      significant figures using the implied REAL*4.
      IF(Y.GT.X)THEN
        WRITE(*,*) 'The register value has been used for Y'
      ELSE
        WRITE(*,*) 'The stored value has been used for Y'
C      /NO_FLOATING_TRACKING or /CHECK has been used.
      ENDIF
      END
```

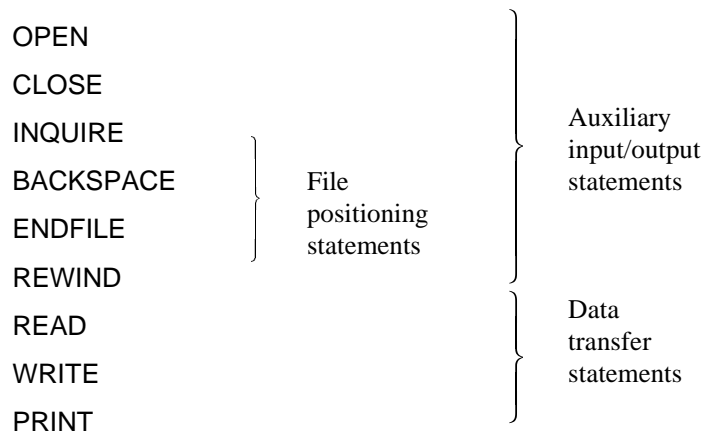

Fortran input/output

Overview

The FTN77 input/output statements allow ANSI standard-conforming programs to be written which can:

- open and close files,
- make inquiries about the type and mode of access of a file,
- access data using any combination of formatted or unformatted and sequential or direct access data transfer statements.

There are nine input/output statements:



Each of these statements has a list of specifiers associated, for example:

```
BACKSPACE      (UNIT=7)
WRITE          (10,REC=9) B,C
```

```
OPEN          (UNIT=3,FILE='FRED')  
READ          (UNIT=4,END=20) A
```

The general form of these specifiers will be obvious from these examples.

A full list of specifiers and the input/output statement(s) to which each applies appears in Table 10-1. Specifiers marked by an asterisk are extensions to the ANSI Standard. Note that the **PRINT** statement has been omitted from this table as specifiers are not permitted.

The input/output statements are described in detail in the remainder of this chapter after the **FTN77** definitions of records, files and methods of access have been explained.

Records

A record is a sequence of characters (for example, a line of text) or a sequence of values (for example, 3 0 4).

There are three kinds of records:

- ☐ Unformatted
- ☐ Formatted
- ☐ Endfile

They are described separately below.

Unformatted record

An unformatted record consists of a series of values in binary form and may contain any combination of numeric, logical or character data or indeed no data at all.

The length of an unformatted record is measured by **FTN77** in bytes and depends on the output list used when it was written. For unformatted sequential files, each record also contains a small amount (usually 2 bytes) of “red tape” which delimit one record from the next (see also page 100).

This means that a file containing (say) 10000 numbers, one per record, will not be as efficient as one composed of 100 records each holding 100 numbers. The size of the individual data items in an unformatted record is shown in Table 10-2.

	WRITE	READ	REWIND	OPEN	INQUIRE	ENDFILE	CLOSE	BACKSPACE
ACCESS				•	•			
BLANK				•	•			
DIRECT					•			
* DRIVER				•				
END		•						
ERR	•	•	•	•	•	•	•	•
EXIST					•			
FILE				•	•			
FMT	•	•						
FORM				•	•			
FORMATTED					•			
NAME					•			
NAMED					•			
NEXTREC					•			
NUMBER					•			
OPENED					•			
REC	•	•						
RECL				•	•			
* RENAME							•	
SEQUENTIAL					•			
SHARE				•				
STATUS				•			•	
UNFORMATTED					•			
UNIT	•	•	•	•	•	•	•	
* NML	•	•						

Table 10-1 Input/output statements and specifiers - allowed combinations are denoted by •

Type	Number of Bytes
INTEGER*1	1
INTEGER*2	2
INTEGER*4	4
REAL*4	4
DOUBLE PRECISION REAL*8	8
COMPLEX*8	8
DOUBLE COMPLEX COMPLEX*16	16
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
CHARACTER*n	n

Table 10-2 length in bytes of unformatted data according to type.

Note:

The length in bytes of a numeric or logical variable depends on the chosen default (/INTS, /INTL, /LOGS, /LOGL and/or /DREAL) for the compilation and also the type statements used for declaration (see page 178).

Formatted record

A formatted record consists of a sequence of characters chosen from the ASCII character set. The *length* of a formatted record is the number of ASCII characters in that record. This length may be zero. The end of the record is indicated by the ASCII character LF (decimal 10).

Endfile record

An endfile record is written by an ENDFILE statement. Such a record may occur only as the last record of a sequential file. In the case of disc files, this record is only a conceptual entity and does not actually exist.

Files

A file is a sequence of records. There are two kinds of files:

- ☐ External
- ☐ Internal

Internal files are described on page 101.

File existence

At any given time there is a set of files that are said to exist for a given program. A file that is known to the operating system may not necessarily exist for a program, for example, because it has been given the “hidden” attribute.

A file may exist for a program and contain no records; an example is a newly created file not yet written.

File names

A file may have a name; if so it is referred to as a named file. Any name, including a pathname, acceptable to the operating system can be used as a file name.

File properties

A file which has been written using formatted, direct access may subsequently be read sequentially and/or examined with a text editor. A special command, **MAKEDA77**, is provided to convert formatted, sequential files into formatted, direct access files, see page 129.

If it is necessary to alter the contents of a direct access formatted file this is possible using an editor, provided **MAKEDA77** is used to reconstruct the result, (remember that editors usually compress files by means of tabs and sometimes do not retain trailing spaces).

Data transferred to or from the screen must be accessed sequentially.

File structure

1. Formatted sequential files

A formatted sequential file consists of zero or more records of the form described on page 98.

2. Formatted direct access files

A formatted direct access file written by FTN77 contains records of the form described on page 98. All records in the file are of the length specified in bytes by the RECL= specifier when the file is opened.

A formatted file created by direct access output may be read by sequential input if desired.

3. Unformatted sequential files

The structure of a record in an unformatted sequential file written by FTN77 is as follows:

- ☐ For the first record or any subsequent record of more than 240 bytes in length:

1-byte	4-bytes	n-bytes	4-bytes	1-byte
FF	nn	xx....xx	nn	FF
indicator byte	record size	record data	record size	indicator byte

- ☐ For other records:

1-byte	n-bytes	1-byte
n	xx....xx	n
record size	record data	record size

This structure is designed to:

- 1) Facilitate very long records
- 2) Minimise the overheads for small records
- 3) Enable backward repositioning (BACKSPACE) without rereading the file.

4. Unformatted direct access files

Each record in an unformatted direct access file contains only the raw data written to it, with no extra “housekeeping” information. All records in the file are of precisely the length specified in bytes by the RECL= specifier when the file is opened.

An attempt to write to a record an amount of data less than the specified record length results in the remainder of the record being undefined. An attempt to write more data than the record can hold results in a run-time error.

File position

A file that is connected to a unit (see page 106) has a position property. The execution of certain input/output statements affects the position of a file. Circumstances such as an error condition can cause the file position to become indeterminate.

The *initial point* of a file is the position immediately before the first record. The *terminal point* is the position immediately after the last record. If a file is positioned within a record, that record is the *current record* otherwise there is no current record.

The terms *preceding record* and *next record* are defined formally in the ANSI Standard; their meanings are self-explanatory for practical purposes.

File access

The two methods of file access are *sequential* and *direct*. The method of access is determined when the file is connected to a unit (see page 106).

An internal file (see below) must be accessed sequentially. Sequential access means that the n 'th record can only be accessed after the preceding $(n-1)$ records have been accessed. Direct access means that the records in a file can be accessed in any order; thus, for example, it is possible to write record 10 even though records 1 to 9 have not been written.

A file that is connected for formatted access may not be accessed by non-formatted data transfer statements for the duration of the connection, and vice-versa.

A file that is connected for direct access may not be accessed by the sequential forms of the data transfer statements for the duration of the connection, and vice-versa.

All records of a direct access file have the same length. Each record in a direct access file is uniquely identified by a positive integer called the *record number* which is specified in the WRITE statement when the record is written. This record number is not stored in the file but once established can never be changed. A direct access record may not be deleted but may be changed by being rewritten.

Internal files

Internal files provide a means of transferring and converting data from internal binary format to character format. Internal files have the following properties:

- 1) The file is a character variable, character array element, character array or character substring.
- 2) A record of an internal file is a character variable, character array element or character substring.
- 3) If the file is not a character array, that is, it is one of the data items in 2) above, it consists of a single record whose length is the same as the length of the character variable, character array element or character substring.
- 4) An internal file record becomes defined:
 - when it is written (if the number of characters written in the record is less than the length of the record, the remaining portion of the record is blank-filled),

- ☐ by means other than by a **WRITE** statement, for example, by a character assignment or **DATA** statement.
- 5) An internal file is always positioned at the start of its first record prior to data transfer. This means that only sequential access is available with internal files.
- 6) List-directed formatting (free format) is not available with internal files according to the **ANSI** Standard. However, **FTN77** does permit this as an extension providing the **/ANSI** compile-time option is not used. For example:

```
CHARACTER*20 NUM
:
:
WRITE(NUM,*) I,J
```
- 7) An auxiliary input/output statement (i.e. **BACKSPACE**, **REWIND**, or **ENDFILE**) can not be used on an internal file.

Units

A *unit specifier* is a means of referring to a screen or a disc file. Units within the range 1 to 100 may exist for a **FTN77** program. All input/output statements (except **CLOSE** and **INQUIRE**) must refer to one of these units.

A unit has the property of being connected or not connected. Once connected, it refers to a file or to a slow peripheral. A unit specifier may become connected in one of the following ways:

- ☐ by execution of the Fortran 77 **OPEN** statement,
- ☐ by preconnection; the standard input and output are preconnected to units 1 and 2 respectively (units 5 and 6 are respective alternatives).

The **ANSI** Standard states that a unit cannot be connected to more than one file at the same time and a file must not be connected to more than one unit at the same time. However, the **OPEN** statement allows the user to change the status of a unit and to connect a unit to a different file. As an extension, **FTN77** allows a file to be opened more than once provided this is done with **STATUS='READONLY'**.

After a unit has been disconnected from a file by the execution of a **CLOSE** statement it may be connected again to the same file or to a different file. After a file has been disconnected from a unit it may be connected to the same unit or to a different unit.

A preconnected unit can be opened in order to redefine its use. Subsequent closing does not re-establish the preconnection.

Unit specifier

All the input/output statements (with the exception of the forms `PRINT*`, and `READ*`) use a unit specifier to refer to a unit. It takes one of the forms

```
UNIT=<unit>
```

```
<unit>
```

where `<unit>` is known as an external unit identifier and is a constant, name or expression of type `INTEGER*4`, `INTEGER*2` or `INTEGER*1`. (`PRINT*` refers to list-directed output on unit 2, which by default is preconnected to the standard output but which may have been reconnected to some other file. Similarly `READ*` refers to list-directed input on unit 1.) The value of `<unit>` must be in the range 1 to 100 for all input/output statements except `CLOSE` and `INQUIRE`. A special form of `<unit>` (an asterisk) is available with the `READ` and `WRITE` statements (see page 122).

If the second form is used, `<unit>` must be the first item in the input/output statement specifier list, for example:

```
WRITE (6) A,B,C
```

This is equivalent to,

```
WRITE (UNIT=6) A,B,C
```

Internal file identifier

An internal file identifier is a character variable, character array element, character array or character substring. It is used in place of an external unit identifier in `READ` or `WRITE` statements when transferring data to or from an internal file, for example:

```
CHARACTER*20 RECORD
. . .
WRITE(RECORD,100) A,B,I
100  FORMAT(2 F8.3,I4)
```

Error and end-of-file conditions

An input/output statement can execute normally or can result in an error condition. The error conditions that can occur are listed in chapter 27. They may also be identified by a suitable call to the `RUNERR@` routine. A `READ` statement can also result in an end-of-file condition. An end-of-file condition (`Iostat` value -1) exists if either of the following events occur:

- an endfile record is encountered during the reading of a file connected for sequential access (in this case the file is positioned after the endfile record),
- an attempt is made to read a record beyond the end of an internal file.

If an error condition occurs during the execution of an input/output statement, execution of the statement terminates and the position of the file becomes indeterminate. It is possible for a program to continue execution after an input/output error has occurred if the `ERR=` and/or `IOSTAT=` specifiers are used (see below).

If an error or end-of-file condition occurs in a `READ` statement, the entities specified in the input list and any implied-`DO variables` become undefined (according to the ANSI Standard), but variables appearing only in subscripts, in substring expressions and in implied-`DO parameters` do not become undefined.

In programs compiled with FTN77, items read in before the error will be available in accordance with the ANSI standard. Similarly, any subscript will have the value it had at the point of the error.

If an error condition occurs during execution of a `WRITE` or `PRINT` statement, any implied-`DO variables` in the output list become undefined.

It is possible to recover from all input/output execution errors by means of the optional `ERR=` and `IOSTAT=` specifiers. Similarly, end-of-file conditions can be dealt with using the `END=` specifier.

The error specifier has the form:

`ERR=<errlab>`

where `<errlab>` is the statement label of an executable statement that appears in the same program unit as the `ERR=` specifier. In the example which follows, control is transferred to the statement labelled 10 if an input error occurs on unit 7:

```
      READ (7, '(3 I5)', ERR=10) I, J, K
      . . .
10    PRINT *, 'data error'
      END
```

The `IOSTAT` specifier has the form:

`IOSTAT=<ios>`

where `<ios>` is an integer variable or integer array element. When an input/output statement containing an `IOSTAT` specifier is executed, the value returned to `<ios>` is as follows:

- zero if neither an error condition nor an end-of-file condition is encountered,

- a positive integer if an error condition is encountered (a library routine, `RUNERR@`, is provided to enable the run-time error corresponding to a given `IOSTAT` value, to be printed on the screen),
- -1 if an end-of-file but no error condition is encountered.

The following program fragment shows the use of `IOSTAT`:

```

      READ (7,100,IOSTAT=I) A,B,X
      IF (I) 1,3,2
1     STOP 'end-of-file reached'
2     IF (I.EQ.84) THEN
      PRINT *, 'No file open on unit 7'
      ELSE
      PRINT *, 'I/O error ', I
      ENDIF
      STOP
3     Y = (A+B)/X
      . . .

```

Note that the statement following the `READ` statement is obeyed in this case no matter what condition is encountered.

The `IOSTAT` specifier can be used in conjunction with the `ERR=` specifier as in the following example:

```

      BACKSPACE (UNIT=10,IOSTAT=I,ERR=50)
      . . .
50    IF (I.GE.200) STOP

```

In this example, if there is an error, the `IOSTAT` variable `I` is given a positive (non-zero) value and control is transferred to the statement labelled 50.

The end-of-file specifier has the form:

`END=<endlab>`

where `<endlab>` is the label of an executable statement that appears in the same program unit as the `END=` specifier. For example:

```

      READ (7,10,END=20) A,B,C
10     FORMAT (3F6.2)
      . . .
20     END

```

Once the end-of-file is reached on unit 7, control is transferred to the statement labelled 20 which in this case is the end of the program. Note the comments on page 98 concerning the form of an endfile record.

Under DOS/Win16, an end-of-file condition is raised when inputting data from the screen by the ASCII character ETX (octal code 203).

Under Win32, an end-of-file condition is raised when inputting data from the screen by the ASCII character Ctrl-Z (decimal 26).

Connecting files

A file may be connected during program execution by means of the **OPEN** statement. It is possible to use the **OPEN** statement to connect a device, such as a standard printer to a unit.

Previously **OPEN**ed files can be disconnected by means of the **CLOSE** statement. The properties of a file (connected or otherwise) can be found by using the **INQUIRE** statement.

The **OPEN** statement

It is possible to open a file dynamically (that is, at run-time) by means of the Fortran 77 **OPEN** statement.

The **OPEN** statement will cause a file to become connected. It is used to describe the properties of a connection in addition to performing the connection itself. For example, in order to open a text file for input, the following statement might appear in a program:

```
OPEN(UNIT=5, FILE='FRED')
```

It will be apparent that the name of the file and a unit number are used together with some defaults provided by the system in order to open the file. **FTN77** implements **OPEN** by calling standard file manipulation routines provided by the operating system.

The general form of the **OPEN** statement is

```
OPEN (<olist>)
```

where <olist> is a list of specifiers:

```
UNIT=<unit>  
IOSTAT=<ios>  
ERR=<errlab>  
FILE=<filename>  
STATUS=<status>  
ACCESS=<access>
```

```

FORM=<form>
RECL=<recl>
BLANK=<blank>
* DRIVER=<driver>
* FILETYPE=<filetype>
* SHARE=<access mode>

```

Note that the specifiers marked * are not in the ANSI Standard and are specific to the FTN77 implementation. As their use is never mandatory, standard-conforming programs can always be compiled and executed.

<olist> must contain exactly one external unit specifier <unit> and may contain, at most, one of each of the other specifiers.

Note that any keyword target (an item enclosed in diamond brackets in the above list of specifiers) of type CHARACTER may be in any combination of upper and lower case characters.

UNIT=<unit>

<unit> is an integer expression (typically a constant or variable) used as an external unit identifier (see page 103).

IOSTAT=<ios>

<ios> is an integer variable or array element which is used as an input/output status specifier (see page 103).

ERR=<errlab>

<errlab> is the label of an executable statement in the current program unit to which control will be transferred in the event of an error (see page 103).

FILE=<filename>

<filename> is a character expression (typically a constant) whose value is a filename or pathname acceptable to the operating system. Note that if the filename part is greater in length than eight characters, then it is truncated. The name may include a suffix, but none is added automatically.

STATUS=<status>

<status> is a character expression (typically a constant), whose value when any trailing blanks are removed is one of the following with the effect described:

'OLD'

A FILE= specifier must also be used and <filename> must exist.

'NEW'

A FILE= specifier must also be used and <filename> must not exist. The file is connected for writing. A sequential file can subsequently be rewound and used for reading.

'SCRATCH'

A FILE= specifier must not be used. A temporary file with the name

(F\$XXXX) will be created where XXXX is a unique 4-digit decimal number between 0000 and 9999 inclusive. When a **STOP** or **END** statement is executed, the file is erased.

'UNKNOWN'

If a **FILE=** specifier is present and <filename> exists, 'UNKNOWN' is equivalent to 'MODIFY'. If **FILE=** is absent, 'UNKNOWN' is equivalent to 'SCRATCH'. If <filename> does not exist 'UNKNOWN' is equivalent to 'NEW'. If **STATUS=** is omitted, 'UNKNOWN' is assumed. Thus, normally, no **STATUS** specifier will be required.

The following options for <status> are not in the ANSI Standard and have been added to the FTN77 implementation:

'APPEND'

A **FILE=** specifier must also be used. 'APPEND' is allowed for both formatted and unformatted files opened for sequential access. Output is appended to <filename> if it exists - if <filename> does not exist, it will be created.

'MODIFY'

A **FILE=** specifier must also be used - <filename> need not exist. If <filename> does not exist, 'MODIFY' is equivalent to 'NEW'. If <filename> exists, 'MODIFY' causes the existing file to be truncated and overwritten (see page 117).

'READONLY'

A **FILE=** specifier must also be used and <filename> must exist. **READONLY** status ensures that any attempt to write a record to <filename> causes a run-time error. It also enables a file to be opened for reading more than once.

ACCESS=<access>

<access> is a character expression (typically a constant), whose value (when any trailing blanks are removed) is either 'SEQUENTIAL' , 'DIRECT' or 'TRANSPARENT' (see the **FORM** specifier below).

ACCESS specifies the method of access for the connection of the file. If the specifier is omitted, 'SEQUENTIAL' is assumed.

FILETYPE=<filetype>

This specifier has been added in the FTN77 implementation in order to give users access to various devices. <filetype> can take the following value with the effect shown:

'TTY'

Input and output is read from the keyboard and written to the screen via the appropriate unit number.

Other devices such as LPT1 may be available by simply opening 'files' of the corresponding name. For example, to write directly to the printer from unit 6 it is often possible to execute:

```
OPEN(UNIT=6, FILE='LPT1')
```

DRIVER=<driver>

<driver> is the name of a subroutine which has been previously declared in an EXTERNAL statement. If DRIVER= appears, then FILE=, FILETYPE= and STATUS= must not appear. DRIVER= is an extension to the ANSI Standard. Its use is described on page 113.

FORM=<form>

<form> is a character expression (typically a constant), whose value is 'FORMATTED', 'UNFORMATTED' or 'PRINTER' when any trailing blanks are removed.

FORM specifies whether the data transfer to and from the program will be formatted, unformatted or in line printer format.

If the specifier is omitted, 'FORMATTED' is assumed if the file is being connected for sequential access and 'UNFORMATTED' is assumed if the file is being connected for direct or transparent access (see the ACCESS specifier above).

FORM='PRINTER' is an extension to the ANSI Standard, and specifies that the first column of any output record is taken as a Fortran carriage control character. The Fortran carriage control characters are as follows:

Character	Vertical Spacing before printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

Carriage return, linefeed, and form feed control characters are output as necessary to give the effects above.

Note that FORM='PRINTER' is only appropriate for files on which output only is performed

For `ACCESS = 'TRANSPARENT'` and `FORM = 'FORMATTED'`, no carriage returns are output at the end of record on output (the user can output carriage returns with the “/” editing descriptor), and on input precisely the field widths specified in the input format are read, with no attempt to align to record boundaries (i.e. after carriage returns).

For `ACCESS = 'TRANSPARENT'` and `FORM = 'UNFORMATTED'`, on output the values in the I/O list are output to file in their internal format, with no surrounding record structure (unlike sequential unformatted). Similarly, on input, the values in the input list are read in direct from file, without any record structure. This gives a Fortran binding for applications which would previously have called the I/O primitive subroutines `OPENR@`, `OPENW@`, `READF@`, `WRITEF@` etc. directly.

RECL=<recl>

This specifier is used when a file is connected for direct access.

As an extension to the standard, `RECL` may also be specified for a file opened for sequential access. This causes fixed length records to be read from or written to file and allows a `BACKSPACE` to be followed by a `WRITE`.

<recl> is an integer expression (typically a constant). It specifies the length of each record in a file being connected for direct access. `RECL` is always measured in bytes.

BLANK=<blank>

`BLANK` must only appear for a file being connected for formatted input/output.

<blank> is a character expression (typically a constant), whose value when any trailing blanks are removed is either `'NULL'` or `'ZERO'`.

If the specifier is omitted, a value of `'NULL'` is assumed.

If `'NULL'` is specified, all blank characters in numeric formatted input fields on the specified unit are ignored except that a field consisting of all blanks has a value of zero. If `'ZERO'` is specified, all blanks, other than leading blanks, are treated as zeros.

SHARE=<access mode>

The operating system provides a means whereby a program, when opening a file, can define the access that other programs are allowed to a file for the period that the first program has the file open. This mechanism is implemented by `SHARE.EXE`, which keeps a track of open files and permits or denies access as appropriate. Thus, in order to use this keyword, you should ensure that `SHARE.EXE` is loaded.

This file sharing mechanism applies for multiple instances of the same file opened by a particular program, for two or more programs running on the same machine

(e.g. in different “DOS boxes” under Windows 3.1 and Windows 95 and different “console windows” under Windows NT), or by two or more programs running on different machines (e.g. access via a shared disk on a network).

When a program opens a file, it can specify that it requires read access, write access, or read and write access. In addition to this, it can specify the <access mode> that other programs are permitted while it still has the file open.

<access mode> is a character expression whose value is one of the following:

'COMPAT'	Compatibility mode - equivalent to opening the file with no sharing attributes. No other program will be able to access the file while this program has it open.
'DENYRW'	Exclusive - no other program can access the file while it is open.
'DENYWR'	Other programs cannot access the file for write or read/write access, but can open the file for read only access.
'DENYRD'	Other programs cannot access the file for read or read/write access, but can open the file for write only access.
'DENYNONE'	Other programs can access the file for read, write or read/write access.

Note that a second or subsequent program attempting to open the file will be denied access in all cases if it attempts to open the file in compatibility mode. All attempts to open a file that may be in use by another program must use one of the other modes, and thereby must specify the access to be granted to other programs trying to access the file subsequently.

It will be seen from the above specification of **OPEN** that there is a large number of possible combinations of specifiers (and defaults). The **OPEN** statement can be used:

- in order to connect an existing file (scratch files or non-existent files are automatically created by **OPEN**),
- in order to connect a user specified device driver (see page 113).

Examples of the use of **OPEN**:

- 1) A file may be connected from within the program as follows:

```
OPEN(5, FILE='DATAFILE', STATUS='OLD')
```

The above statement could be written to include the **UNIT=** specifier thus:

```
OPEN(UNIT=5, FILE='DATAFILE', STATUS='OLD')
```

In this example the file 'DATAFILE' must exist.

- 2) If blanks in numeric fields were to be treated as zeros, the **OPEN** statement in Example 1 would be written as follows:

```
OPEN(5, FILE='DATAFILE', BLANK='ZERO', STATUS='OLD')
```

Note: The **BLANK=** specifier must specify **'ZERO'** when running a Fortran 66 program whose data contains significant blanks in numeric fields.

- 3) As a result of the following **OPEN** statement:

```
OPEN(4, FILE='OUTPUT', STATUS='NEW')
```

unit 4 would be connected to a previously non-existent file called **OUTPUT**. The details of the connection would be as follows:

```
STATUS = 'NEW'  
ACCESS = 'SEQUENTIAL'  
FORM = 'FORMATTED'
```

- 4) If a program containing the **OPEN** statement from example 3) were subsequently rerun without first deleting **OUTPUT**, a run-time failure would result as the use of **STATUS='NEW'** means that the file must not exist.

The **STATUS='UNKNOWN'** keyword avoids this problem, as in the following example:

```
OPEN(4, FILE='OUTPUT', STATUS='UNKNOWN')
```

If **OUTPUT** does not exist, the above **OPEN** statement is identical to that in example 3). If, however, **OUTPUT** exists (perhaps as the result of a previous run), it will be emptied if the first operation is a **WRITE**. If **STATUS** is omitted completely, the default value **'UNKNOWN'** is assumed.

- 5) If **OUTPUT** does not exist, the following statement is equivalent to the **OPEN** statement in example 3):

```
OPEN(4, FILE='OUTPUT', STATUS='APPEND')
```

If **OUTPUT** does exist, any output will be appended and its previously existing contents left unchanged.

Notes:

- ☐ **STATUS='APPEND'** is an FTN77 extension to the ANSI standard.
- ☐ If the specified file is subsequently rewound it will be positioned at record 1, that is, all the records in the file will be available for reading.

- 6) The following **OPEN** statement would create a scratch file for use only during the program run:

```
OPEN(3, STATUS='SCRATCH')
```

A temporary file would be created and erased at the end of the program run. The properties of the connection would be those listed in example 3).

- 7) In order to open a file called **RANDOM** for direct access, the following **OPEN** statement should be used:

```
OPEN(5, FILE='RANDOM', ACCESS='DIRECT',
+ STATUS='OLD', RECL=20)
```

Notice the default for **FORM** is '**UNFORMATTED**' when the statement **ACCESS='DIRECT'** is specified. The connection would establish the following properties:

```
STATUS='OLD'
ACCESS='DIRECT'
FORM='UNFORMATTED'
RECL= 20 (bytes)
```

In other words, the file would be used for direct access without any compression of blanks taking place.

- 8) In order to open a file called **MIXUP** for formatted direct access, the following **OPEN** statement should be used:

```
OPEN(6, FILE='MIXUP', ACCESS='DIRECT',
+ STATUS='MODIFY', FORM='FORMATTED', RECL=80)
```

The record length for a formatted direct access file is specified in characters. The properties established by the connection would be as follows:

```
STATUS='MODIFY'
ACCESS='DIRECT'
FORM='FORMATTED'
RECL= 80 (bytes)
```

- 9) In order to open a scratch file for unformatted, sequential access:

```
OPEN(3, STATUS='SCRATCH',
+ FORM='UNFORMATTED', ACCESS='SEQUENTIAL')
```

A scratch direct access file could be **OPENed** as follows:

```
OPEN(4, STATUS='SCRATCH', ACCESS='DIRECT', RECL=64)
```

User-supplied input/output device drivers

The **DRIVER=** keyword has been provided for use with the **OPEN** statement in order to allow the use of user-supplied device drivers. This facility means that all the Fortran 77 input/output statements can be used to refer to a “device” such as a non-

standard printer, plotter etc., as well as providing facilities such as output to the screen and file simultaneously.

The facility can be used with any combination of formatted or unformatted, sequential or direct access input and output. Any Fortran unit can be connected to a device driver as in the following example:

```
EXTERNAL MYDEV
OPEN(5, DRIVER=MYDEV)
```

Here MYDEV is a user supplied subroutine which will handle formatted, sequential input or output on unit 5. In general, the target of the DRIVER= keyword is the name of a subroutine with the following specification:

```
SUBROUTINE driver_name(BUFF, BSIZE, BLEN, ACTION,
+                      IFAIL)
  INTEGER*2 BSIZE, BUFF(BSIZE), BLEN, ACTION, IFAIL
```

ACTION is specified in Table 10-3.

IFAIL is set to zero when the user-supplied routine is called by the input/output system. If IFAIL is given a non-zero value by the driver routine, the FTN77 error trapping mechanism will be invoked on exit from that routine. (Thus, depending on whether the IOSTAT= keyword has been used in the corresponding Fortran input/output statement, the run-time traceback mechanism will be invoked or the program can take action for a non-zero IOSTAT value.)

Value of ACTION	Corresponding Fortran statement
1	FORMATTED SEQUENTIAL READ
2	FORMATTED SEQUENTIAL WRITE
3	ENDFILE
4	REWIND
5	BACKSPACE
6	OPEN
7	CLOSE
8	INQUIRE
9	UNFORMATTED SEQUENTIAL READ
10	UNFORMATTED SEQUENTIAL WRITE
11	FORMATTED DIRECT ACCESS READ
12	FORMATTED DIRECT ACCESS WRITE
13	UNFORMATTED DIRECT ACCESS READ
14	UNFORMATTED DIRECT ACCESS WRITE

Table 10-3, Effect of ACTION in User-supplied input/output, device drivers

The **IOSTAT** value returned for a non-zero **IFAIL** value is always 155. In order to obtain a more specific error message on error exit from a driver routine, a common block variable should be used to return an appropriate error code to the calling program unit.

Note that it is not possible to perform Fortran input/output in a driver routine.

BUFF, **BSIZE** and **BLen** are only relevant for values of **ACTION** corresponding to the **READ** and **WRITE** statements. **BUFF(BSIZE)** is an array which is used as a buffer. It holds either two characters per **INTEGER*2** word for formatted input/output or is used to hold binary information for unformatted input/output.

The value of **BLen** depends on whether the driver routine is being used for input or for output as follows:

- **READ** statement (**ACTION** = 1). The driver routine must set **BLen** as follows:

Formatted input: **BLen** is the number of characters that have been input.

Unformatted input: **BLen** is the number of bytes that have been input.

- **WRITE** statement (**ACTION** = 2). The value of **BLen** is set on entry to the driver routine as follows:

Formatted output: **BLen** is the number of characters to be output. Note that a **CR** or **LF** character is not added by the input/output system prior to entry to the driver routine.

Unformatted output: **BLen** is the number of bytes to be output.

It is recommended that driver routines are written in the following manner to ensure that all possible values of **ACTION** are catered for in a program:

```

SUBROUTINE MYDRIV(B,NB,NCH,ACTION,IFAIL)
  INTEGER*2 B(NB),NCH,ACTION,IFAIL
C
C   Use a computed GOTO to allow for all values of
C   ACTION even though we only expect the routine to
C   be entered for the OPEN, READ and WRITE statements
C
      GOTO (1,2,3,3,3,4,3,3,3,3,3,3,3,3),ACTION
C   Error exit
3    IFAIL = 999
      RETURN
C   READ
1    . . . . .
      RETURN
C   WRITE
2    . . . . .
      RETURN
```

```
C OPEN (NEEDED AS THE DRIVER ROUTINE IS CALLED
C IMMEDIATELY FROM THE Fortran 77 OPEN STATEMENT WHEN
C DRIVER= IS USED)
4 RETURN
END
```

The CLOSE statement

The general form of the CLOSE statement is:

CLOSE (<clist>)

where <clist> is a list of specifiers:

UNIT=<unit>
IOSTAT=<ios>
ERR=<errlab>
STATUS=<status>
RENAME=<newname>

Note:

The RENAME specifier is not in the ANSI Standard and has been added to the FTN77 implementation.

The variables or array elements represented by <unit>, <ios> and <status> may be of type INTEGER*4, INTEGER*2 or INTEGER*1.

Notes:

- It is possible to execute a CLOSE statement that specifies a unit that neither exists nor has a file connected to it. Use of such a CLOSE statement has no effect.
- All files are automatically closed by the input/output system when a program terminates.

UNIT=<unit>

<unit> is an external unit identifier (see page 103). If the value of <unit> is less than or equal to zero, CLOSE produces a run-time error.

IOSTAT=<ios>

<ios> is an input/output status specifier (see page 103) which must be an integer variable or array element.

ERR=<errlab>

<errlab> is the label of an executable statement in the current program unit to which control will be transferred in the event of an error (see page 103).

STATUS=<status>

<status> is a character expression (typically a constant) whose value when any trailing spaces are removed is 'KEEP' or 'DELETE'. The character expression may comprise any combination of upper and lower case characters.

If the **STATUS** specifier is omitted, the assumed value is 'KEEP' for a named file or 'DELETE' for a scratch file.

If 'KEEP' is specified or assumed, the file continues to exist after the **CLOSE** statement has been executed.

If 'DELETE' is specified or assumed, the file is erased by the **CLOSE** statement.

RENAME=<newname>

<newname> is a character expression (typically a constant) which must represent a pathname. This specifier permits the file opened on <unit> (which may be a scratch file) to be renamed on being closed. No error is issued if **STATUS**='DELETE'.

Example:

```
CLOSE (UNIT=4, STATUS='DELETE')
```

would close the file currently connected to unit 4. All trace of the file would be removed from the system.

The INQUIRE statement

INQUIRE allows the user to find out the properties of a particular named file or of the connection or availability of a particular unit. The **INQUIRE** statement may be executed before, while or after a file is connected to a unit. All values assigned by the **INQUIRE** statement are those that are current at the time the statement is executed.

Note that all value assignments are done in accordance with the rules for assignment statements so in the case of character information, truncation or padding with blanks will occur. This can be used to advantage, for example:

```
CHARACTER*11 F
INQUIRE (UNIT=6, FORM=F)
```

would return **F** with either the value 'FORMATTED' or the value 'UNFORMATTED'. If **F** were declared as

```
CHARACTER F
```

its value would be returned as either 'F' or 'U'.

The **INQUIRE** statement takes one of the following forms:

Inquire by file: **INQUIRE** (**FILE**=<filename>, <inqlist>)

Inquire by unit: **INQUIRE** (**UNIT**=<unit>, <inqlist>)

where <filename> is a character expression whose value when trailing blanks are removed is a filename (acceptable to the operating sytem) which is the subject of the inquiry. The named file need not exist or be connected to a unit. Note that a file may be referred to by pathname.

<unit> is an external unit identifier (see page 103) of type either INTEGER*1, INTEGER*2 or INTEGER*4. The specified unit need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and the file connected. <inqlist> is a list of specifiers chosen from:

```
IOSTAT=<ios>
ERR=<errlab>
EXIST=<exist>
OPENED=<opened>
NUMBER=<number>
NAMED=<named>
NAME=<pname>
ACCESS=<access>
SEQUENTIAL=<seq>
DIRECT=<dir>
FORM=<form>
FORMATTED=<fmt>
UNFORMATTED=<unf>
RECL=<recl>
NEXTREC=<next>
BLANK=<blank>
FUNIT=<filehandle>
```

A variable or array element that may become defined or undefined as a result of its use as a specifier in an INQUIRE statement must not be referenced by any other specifier in the same INQUIRE statement.

If no error condition occurs in either an INQUIRE by file or INQUIRE by unit statement, <exist> and <opened> always become defined. If an error condition occurs during the execution of either type of INQUIRE statement, all of the specifier variables and array elements except <ios> become undefined.

Execution of an INQUIRE by file statement causes the specified variables or array elements <named>, <pname>, <seq>, <dir>, <fmt> and <unf> to be assigned values only if the value of <filename> is acceptable to the operating sytem as a file name and if the specified file exists. The specified variables <number>, <access>, <form>, <recl>, <next>, <funit> and <blank> become defined only if <opened> becomes defined with the value .TRUE.

Execution of the INQUIRE by unit statement causes the specified variables or array elements <number>, <named>, <pname>, <access>, <seq>, <dir>, <form>, <fmt>, <funit>, <blank> and <ios> to be assigned values only if the unit is connected to a file.

<unf>, <recl>, <next>, <blank> and <funit> to be assigned values only if the specified unit exists and is connected to a file.

The variables or array elements represented by <ios>, <number>, <recl>, <next> may be of type INTEGER*4, INTEGER*2 or INTEGER*1. Similarly, the variables or array elements represented by <exist>, <opened> and <named> may be of type LOGICAL*4, LOGICAL*2 or LOGICAL*1.

A full description of the list of specifiers follows below:

IOSTAT=<ios>

<ios> is an input/output status specifier (see page 103).

ERR=<errlab>

<errlab> is the label of a statement to which control is to be transferred in the event of an error (see page 103).

EXIST=<exist>

<exist> is a logical variable or logical array element.

Execution of an INQUIRE by file statement causes <exist> to be assigned the value .TRUE. if there exists a file with the specified name; if not, <exist> is assigned the value .FALSE.

Execution of an INQUIRE by unit statement causes <exist> to be assigned the value .TRUE. if the specified unit exists; if not, <exist> is assigned the value .FALSE.

OPENED=<opened>

<opened> is a logical variable or logical array element.

Execution of an INQUIRE by file statement causes <opened> to be assigned the value .TRUE. if the file specified is connected to a unit; if not, <opened> is assigned the value .FALSE.

Execution of an INQUIRE by unit statement causes <opened> to be assigned the value .TRUE. if the specified unit is connected to a file; if not, <opened> is assigned the value .FALSE.

NUMBER=<number>

<number> is an integer variable or integer array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, <number> becomes undefined.

NAMED=<named>

<named> is a logical variable or logical array element that is assigned the value .TRUE. if the file has a name and .FALSE. otherwise.

NAME=<pname>

<pname> is either a character variable or character array element that is assigned the value of the (path)name of the file, if the file has a name; if not, it becomes

undefined. If this specifier appears in an **INQUIRE** by file statement, its value need not be the same as the name given in the **FILE=** specifier.

ACCESS=<access>

<access> is a character variable or character array element that is assigned the value 'SEQUENTIAL', 'DIRECT' or 'TRANSPARENT' depending on the current mode of access. If there is no connection, <access> becomes undefined.

SEQUENTIAL=<seq>

<seq> is a character variable or character array element that is assigned the value 'YES' if SEQUENTIAL is included in the set of allowed access methods for the file, 'NO' if SEQUENTIAL is not included in the set of allowed access methods for the file, and 'UNKNOWN' if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

DIRECT=<dir>

<dir> is a character variable or character array element that is assigned the value 'YES' if DIRECT is included in the set of allowed access methods for the file, 'NO' if DIRECT is not included in the set of allowed access methods for the file, and 'UNKNOWN' if the processor is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

FORM=<form>

<form> is a character variable or character array element that is assigned the value 'FORMATTED' if the file is connected for formatted input/output, and is assigned the value 'UNFORMATTED' if the file is connected for unformatted input/output. If there is no connection, <form> becomes undefined.

FORMATTED=<fmt>

<fmt> is a character variable or character array element that is assigned the value 'YES' if FORMATTED is included in the set of allowed forms for the file, 'NO' if FORMATTED is not included in the set of allowed forms for the file, and 'UNKNOWN' if the processor is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

UNFORMATTED=<unf>

<unf> is a character variable or character array element that is assigned the value 'YES' if UNFORMATTED is included in the set of allowed forms for the file, 'NO' if UNFORMATTED is not included in the set of allowed forms for the file, and 'UNKNOWN' if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

RECL=<recl>

<recl> is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. The length is measured in bytes. If there is no connection or if the connection is not for direct access, <recl> becomes undefined.

NEXTREC=<next>

<next> is an integer variable or integer array element that is assigned the value $n+1$, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, <next> is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, <next> becomes undefined.

BLANK=<blank>

<blank> is a character variable or character array element that is assigned the value 'NULL' if null blank control is in effect for the file connected for formatted input/output, and is assigned the value 'ZERO' if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, <blank> becomes undefined.

FUINT=<filehandle>

<filehandle> is an integer variable that is assigned to the internal file handle of an already opened file (for DBOS and Win16 ClearWin+ applications this is also the system file handle). This value can be used in calls to the Salford library routines READF@, WRITEF@, WRITEFA@, FPOS@ and RFPOS@ but should not be used with CLOSEF@.

Examples:

- INQUIRE by file. In order to find out if a file called HISFILE exists:

```

      CHARACTER*1 SEQ,FMT
      LOGICAL EX,OPND
      . . .
      INQUIRE (FILE='HISFILE',NUMBER=N,EXIST=EX,
+ OPENED=OPND,SEQUENTIAL=SEQ,FORMATTED=FMT)
C   ensure the file exists and has not been opened
      IF (EX.AND..NOT.OPND) THEN
C   if it does, and it has not been opened,
C   and its properties are suitable, open it
C   for formatted, sequential access (using defaults)
      IF (SEQ.EQ.'Y'.AND.FMT.EQ.'Y') THEN
          OPEN (UNIT=4,FILE='HISFILE')
      ENDIF
    ENDIF
  . . .

```

- INQUIRE by unit. To find the name of a file connected to a unit:

```

      CHARACTER*20 FNAME
      LOGICAL OPND,NAMED
      . . .

```

```
      N = 4
      . . .
      INQUIRE (UNIT=N, NAME=FNAME, NAMED=NAMED, OPENED=OPND)
      IF (OPND) THEN
        IF (NAMED) THEN
          PRINT *, FNAME, ' connected to unit ', N
        ELSE
          PRINT *, 'unnamed file connected to unit ', N
        ENDIF
      ELSE
        PRINT *, 'UNIT ', N, ' not connected'
      ENDIF
      . . .
```

Data transfer statements

This section gives the formal definitions of the data transfer statements **READ**, **WRITE** and **PRINT**. Their use is explained more fully on pages 128 to 130. The statements have the following general form, where items in square brackets are optional :

```
      READ ( <cilist> ) [ <iolist> ]
      READ <fmt> [, <iolist> ]
      WRITE ( <cilist> ) [ <iolist> ]
      PRINT <fmt> [, <iolist> ]
```

where

<iolist> is an input/output list (not described further).

<fmt> is a format identifier (see **FMT=<fmt>** in the description of <cilist> below).

<cilist> is a control information list chosen from the following:

```
      UNIT=<unit>
      FMT=<fmt>
      NML=<nlist>
      IOSTAT=<ios>
      ERR=<errlab>
      END=<endlab>      (only allowed with READ)
      REC=<recno>
```

A control information list must contain a **UNIT** specifier. The remaining specifiers are optional.

UNIT=<unit>

<unit> is one of the following:

- ☐ An integer constant, variable or expression with a value that is an external unit identifier (see page 103).
- ☐ A character variable, array name or array element name, or a substring that is an internal file identifier (see page 103).
- ☐ An asterisk, denoting an external unit provided by the system. The FTN77 implementation uses 1 or 5 for input and 2 or 6 for output.

If the optional characters **UNIT=** are omitted, the specifier must be the first one in <cilist>.

FMT=<fmt>

<fmt> is format identifier and must be one of the following:

- ☐ The statement label of a **FORMAT** statement that appears in the same program unit as the format identifier.
- ☐ An integer variable name that has been **ASSIGNED** the statement label of a **FORMAT** statement that appears in the same program unit as the format identifier.
- ☐ An asterisk, specifying list-directed formatting. The default edit descriptors for list-directed output for the different variable types are as follows:

Integer	Logical	Real	Double Precision
I12	L3	1PG16.6	1PD27.12

Note that complex and double complex variables are equivalent to a pair of variables of type real or double precision respectively and are output, separated by a comma, and enclosed in parentheses.

- ☐ A variable, array name, constant or expression of type character.
- ☐ A non-character array name (only if the compile-time option **/ANSI** is not in use).

In cases where the format specification is decoded at run-time, inner format specifications may not be nested beyond a depth of 10. In other cases, FTN77 imposes a nesting limit of 20.

If the optional characters **FMT=** are omitted from this specifier, the format identifier must be the second item in <cilist> and the first item must be the unit specifier without the optional characters **UNIT=**. In other words, the statement:

```
WRITE (3, 100) X
```

is valid Fortran 77.

NML=<nlist>

Namelist-directed I/O is a common Fortran 77 extension which is implemented in FTN77. It provides a powerful mechanism for input and output of variables, associating values directly with variable names. Namelist-directed I/O can be performed for any file connected for formatted sequential access. Formatted, list-directed, and namelist-directed I/O can be freely mixed for a file connected for formatted sequential access, although the syntax of the input record must always be suitable for the intended operation.

The first element in using namelist-directed I/O is to define a namelist. This is done by a statement syntactically similar to a **COMMON** statement, associating a group of variables with a particular namelist name:

NAMELIST / <namelist-name> / <variable-list>

for example:

NAMELIST / DIMENSIONS / HEIGHT, WIDTH, DEPTH

A particular variable may belong to more than one namelist within a program unit. The namelist name is local to the program unit in which the **NAMELIST** statement appears. The name used for the namelist must not be used for any other object in the program unit in question.

A **NAMELIST** statement specifying a namelist which has already appeared in an earlier namelist statement specifies that the variables specified therein are to be appended to the namelist in question. In this respect the behaviour of the namelist statement is analogous to that of the **COMMON** statement, where subsequent **COMMON** statements for the same common block append the specified variables to the common block.

Namelist-directed input is accomplished by specifying the namelist name as part of the control list for the read statement. No I/O list is either required or permitted. Thus, a namelist-directed **READ** for the example above could be:

READ(7,NML=DIMENSIONS)

or

READ(7,DIMENSIONS,END=99)

Note in the second case that the namelist name can be used in the position that a format specifier would be used for a formatted I/O transfer. The "END=" and "IOSTAT=" specifiers can be used as normal.

The rules for the form of the input record are as follows:

- The record must begin with the namelist name preceded by an ampersand ("&")

- ❑ Input is then taken in the form “<variable> = <value>” for variables specified as members of the namelist in question in the corresponding NAMELIST statement.
- ❑ The record is terminated by a slash character (“/”).
- ❑ Variables do not have to appear in the same order within the input record as they appear in the NAMELIST statement.
- ❑ Every variable in the NAMELIST statement does not have to appear in the input record. Variables which do not appear in the input record are left with their values unmodified.
- ❑ Variables can appear more than once in the input record. Value assignments take place in the order they appear in the input record, so that assignments appearing later in the record take precedence.
- ❑ Values for character variables must be delimited by apostrophes, as for list-directed I/O.
- ❑ The form of the input record is quite flexible, with spaces allowed where they do not break up the namelist name, variable names etc. The record itself can be split across several lines, so the term “record” here is used somewhat differently from the case for standard formatted sequential I/O.

Thus, a suitable input record for the READ statement above could be:

```
&DIMENSIONS
  DEPTH = 12.4
  HEIGHT = 16.5
/
```

Note that WIDTH does not appear in the above, and would be left unmodified.

When a namelist-directed READ statement is executed, the input file is scanned for a suitable input record, and all input is discarded until one is found. Thus, if you mis-spell the namelist name in the input record, the likely symptom of this will be that the namelist-directed READ will encounter end-of-file as it reads past the intended record.

Arrays can be read either by specifying a particular element, or by supplying values for all the elements in the array. Assignment to arrays is performed in row-major order. Multiple values can be specified by means of a repeat count, for example “10*3”, and values can be skipped over by specifying a “null” value in that particular position in the list of values. A repeat count can also be applied to a null value.

For example, consider:

```
INTEGER IARR(10)
NAMELIST / EXAMPLE / IARR
```

```
DATA IARR / 10*0 /  
.  
.  
.  
READ (8, EXAMPLE)
```

with the input record:

```
&EXAMPLE  
IARR = 1, 2, , 4, 3*5, 2*, 10  
IARR(7) = 7  
/
```

This would result in the array **IARR** taking the values:

```
1, 2, 0, 4, 5, 5, 7, 0, 0, 10
```

Note that the initial assignment to **IARR(7)** is overridden by a later assignment specifically to that element.

The usual substring notation can be applied to character variables in the input record. For example:

```
&EXAMPLE2  
CH(2:3) = 'XX'  
/
```

Where **CH** is a member of the namelist **EXAMPLE2**.

For namelist-directed output, a record is output in a syntax similar to that described above, with the values of all variables in the namelist output in the order the variables appear in the namelist declaration. For example, for the namelist **DIMENSIONS** given above, the namelist-directed output statement:

```
WRITE (9,DIMENSIONS)
```

might produce the output record:

```
&DIMENSIONS  
HEIGHT = 12.3000  
WIDTH = 16.2000  
DEPTH = 13.3000  
/
```

IOSTAT=<ios>

<ios> is either an integer variable or array element used as an input/output status specifier (see page 103).

ERR=<errlab>

<errlab> is the label of a statement to which control is to be transferred in the event of an input/output error (see page 103).

END=<endlab>

(Only allowed with the READ statement.)

<endlab> is the label of a statement to which control is to be transferred in the event of an end-of-file condition (see page 103). <endlab> can, of course, be the same as <errlab>.

REC=<recno>

<recno> is an integer variable, constant or expression which specifies the number of a record that is to be read or written in a file connected for direct access.

If <cilist> contains a format specifier (FMT=), the statement is a formatted input output statement; if not, it is an unformatted input/output statement.

If <cilist> contains a record specifier (REC=), the statement is a direct access input/output statement; if not, it is a sequential input/output statement.

It is important to note that the first WRITE statement for a file opened for sequential access will remove any previous contents unless STATUS= 'APPEND' was used when the file was opened.

Clearly, there are four combinations of file access that are specified by the standard:

1) **FORMATTED, SEQUENTIAL**

This is available for all disc files and sequentially accessed peripherals. It is the method of access used for example to read a text file and to print lines of text.

It is the only file access method allowed by the PRINT statement and the simple form of the READ statement (i.e. READ*,). Files are written without the DOS end of file marker (Ctrl-Z). Files containing Ctrl-Z will be read correctly but the marker will be ignored. On reading, tabs are replaced by spaces.

2) **UNFORMATTED, SEQUENTIAL**

This is the traditional means of intermediate output for large programs.

3) **FORMATTED, DIRECT**

It is possible to use direct access to read a file of text that could have been created, for example, by typing at the keyboard using an editor.

Files are written without the DOS end of file marker (Ctrl-Z) and will cause an error on reading if they contain this marker.

Files created other than by an FTN77 program must, however, be converted to fixed length uncompressed records by means of the MAKEDA77 command (see page 130).

4) **UNFORMATTED, DIRECT**

These files are created (written) without any 'red tape' information and should, therefore, be readable by non-FTN77 programs. FTN77 can read any file created in this 'raw' form, provided that the file length is an exact multiple of the record length.

The remainder of this section describes the various forms of the data transfer statements in more detail, together with examples where appropriate.

Formatted, sequential access

A file must be connected to a unit and/or **OPEN**ed before an input/output statement can refer to that unit.

When the **OPEN** statement is used, the **ACCESS** and **FORM** specifiers must either be omitted (defaults used) or specified as:

```
ACCESS='SEQUENTIAL'  
FORM='FORMATTED'
```

Examples:

- 1) The following statements:

```
OPEN(4,FILE='FRED',STATUS='NEW')  
. . .  
WRITE(4,'(F12.3)') X
```

would write the value of **X** to file **FRED** opened on unit 4.

- 2) The statements:

```
OPEN(7,FILE='OLDFILE')  
. . .  
READ(7,100) A,B,C  
100 FORMAT(3 F10.2)
```

would read three values from an existing file opened on unit 7.

- 3) The program fragment:

```
N = 7  
ASSIGN 10 TO IFORM  
10 FORMAT (3 I3)  
. . .  
READ(N,IFORM) I,J,K
```

would read three values from unit 7 using the **FORMAT** statement labelled 10.

- 4) The statement:

```
PRINT *, X,Y,Z
```

would use list-directed formatting to output three real values to unit 2 (the external unit used by **PRINT**). The statement:

```
WRITE (*,*) X,Y,Z
```

would have the identical effect. The first asterisk is the unit number provided by the system for formatted output (unit 2) and the second asterisk implies list-directed formatting.

The statements:

```
CHARACTER*20 X
. . .
Y = 4.321417
WRITE(X, '(F20.6)') Y
```

would use `X` as an internal file. The value of `Y` would be converted to character form in `X` which could then be used, for example, in a character assignment.

Unformatted, sequential access

Files are made available to the program by the `OPEN` statement.

The `OPEN` statement should specify `ACCESS='SEQUENTIAL'` (or use this default) and `FORM='UNFORMATTED'`.

Example:

Unformatted output to file may be achieved as follows:

```
DIMENSION A(30)
OPEN (3, FILE='UNFILE', STATUS='NEW', FORM='UNFORMATTED')
. . .
WRITE (3) A
```

As a result, array `A` would be written to the disc file `UNFILE`.

Formatted, direct access

Files created in this way may be listed in the ordinary way. A file required for direct access can be `OPENed`.

If the `OPEN` statement is used, `ACCESS='DIRECT'` and `FORM='FORMATTED'` must be specified. For example:

```
INTEGER PARTNO
CHARACTER*80 RECORD,ANS*1
OPEN(1, FILETYPE='TTY')
C open file of card images:
  OPEN(UNIT=3, FILE='STOCKLIST', ACCESS='DIRECT',
+     FORM='FORMATTED', RECL=80)
C read part number:
1  READ (1, '(I3)') PARTNO
  IF (PARTNO.LE.0) STOP
C read details from stocklist and print:
```

```
      READ(3,'(A)',REC=PARTNO,ERR=3) RECORD
      WRITE(2,'(''PART'',I3,1X,A)') PARTNO,RECORD
      PRINT *,'update required? answer Y or N'
2     READ(1,'(A)') ANS
      IF (ANS.EQ.'Y') THEN
          PRINT *,'type updated record'
          READ(1,'(A)') RECORD
C     write updated record to direct access file:
          WRITE(3,'(A)',REC=PARTNO) RECORD
          ELSEIF(ANS.EQ.'N') THEN
              STOP
          ELSE
              PRINT *,'answer Y or N'
              GOTO 2
          ENDIF
          GOTO 1
3     WRITE (2,'('' INVALID PART NUMBER '',I3)') PARTNO
      GOTO 1
      END
```

This simple program would read a part number from the keyboard (unit 1) and use it to access a record of a direct access file opened on unit 3. This record would then be printed and the user would be asked whether the record was to be updated. If the answer were yes, the updated record would be typed and it would be written to the direct access file. This file could be listed after the program run was complete.

Note that the **MAKEDA77** command should be used if a user wishes to change a formatted file accessed sequentially into one capable of being used for direct access. The following sequence indicates the interactive nature of the command (user responses are printed in *italic type*):

```
MAKEDA77
Enter desired record length: <n>
Enter sequential source pathname: <input pathname>
Enter direct access output pathname: <output pathname>
```

The record length in characters, <n>, must be a positive integer less than 2001. The input and output pathnames are truncated to the first 100 characters (filenames may, of course, be specified instead of pathnames).

Unformatted, direct access

An unformatted direct access file can be made available to a program by the **OPEN** statement.

If the **OPEN** statement is used, the statement **ACCESS='DIRECT'** must be specified (note that **FORM='UNFORMATTED'** is the default for direct file access).

Example:

```

      DIMENSION A(60)
      OPEN(UNIT=7,FILE='INPUT',STATUS='OLD',
+        FORM='UNFORMATTED')
      OPEN(UNIT=8,FILE='OUTPUT',STATUS='MODIFY',
+        ACCESS='DIRECT',RECL=128)
      N = 1
1     READ (7,END=3)A
      WRITE(8,REC=N,ERR=2)A
      N = N + 1
      GOTO 1
2     PRINT *, 'error writing file'
3     END

```

The above program would read records from a sequentially written file in order to create a direct access file.

File positioning statements

The forms of the file positioning statements are:

```

      BACKSPACE <unit>
      BACKSPACE (<list>)
      ENDFILE <unit>
      ENDFILE (<list>)
      REWIND <unit>
      REWIND (<list>)

```

where <unit> is an external unit identifier (see page 103). <list> is a list of specifiers as follows:

```

      UNIT=<unit>
      IOSTAT=<ios>
      ERR=<errlab>

```

where <unit>, <ios> and <errlab> are as described previously. Note that a UNIT= specifier must be present.

A BACKSPACE, ENDFILE or REWIND statement may refer to any disc file open for sequential access. These statements may also refer to sequential input/output read from the keyboard or written to the screen.

BACKSPACE statement

Execution of a **BACKSPACE** statement causes the disc file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is unchanged. If the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing over records written using list-directed formatting is prohibited.

ENDFILE statement

Execution of an **ENDFILE** statement writes an endfile record as the next record of an disc file.

The file is then positioned after the endfile record. After execution of an **ENDFILE** statement, either a **BACKSPACE** or **REWIND** statement, as appropriate, must be used to reposition the file prior to execution of any data transfer input/output statement.

An endfile record for a file is in fact a 'dummy' record which enables the Fortran 77 input/output system to detect that a file is positioned at end-of-file so that a run-time error can be produced if an attempt is made to write to that file before it has been backspaced or rewound. Nothing is actually written to the file as a result of an **ENDFILE** statement (except that file truncation may occur if the file is currently positioned somewhere other than at the end) and thus the statement is only of use when writing portable Fortran programs which might run on a system where endfile records are physically read and written.

REWIND statement

Execution of a **REWIND** statement causes the specified disc file pointer to be positioned at its initial point. If **STATUS='APPEND'** is used in the **OPEN** statement which connected the file, the initial point is not the point at which data had been added should the file not originally have been empty.

If a file is already positioned at its initial point, the **REWIND** statement has no effect.

Extensions to the standard

This section summarises the FTN77 input/output extensions. They are all available by default but some extensions may cause a compile-time warning. Use of the **/ANSI** compile-time option will cause all the extensions described here to cause either a compile-time or a run-time error.

Extensions to the OPEN Statement

The OPEN statement has three extra options to the STATUS= specifier, 'APPEND', 'MODIFY' and 'READONLY', together with the extra specifiers FILETYPE= and DRIVER=.

The DRIVER= keyword is provided for use with the OPEN statement in order to allow the specification of user-supplied device driver routines for both formatted and unformatted input/output.

The RECL= specifier may be specified for files to be accessed sequentially. This allows unrestricted use of the BACKSPACE statement in conjunction with WRITE.

These extensions are fully described on page 106.

Extensions to the CLOSE Statement

The CLOSE statement has an extra specifier, RENAME=. This extension is fully described on page 116.

Input/output of binary, octal and hex. values

FTN77 provides the extra edit descriptors Ow.m, Zw.m and Bw.m to facilitate the use of octal, hexadecimal and binary values for input and output. These descriptors are described on page 183.

The handling of list-directed input has been extended to support octal, hexadecimal and binary values.

The following list indicates the various number formats for decimal -1 :

Octal	Hexadecimal	Binary
O'17777'	Z'FFFF'	B'1111111111111111'

In addition, character data may be read into non-character variables either in list-directed or non list-directed input mode.

Business Editing

Business editing is intended for accounting programs in which the following features are desirable:

- ☐ Filling of number fields, thus preventing subsequent modification, for example when printing cheques.
- ☐ Suppression of leading zeros and plus signs.
- ☐ Printing of trailing minus signs (accounting convention).
- ☐ Conversion of trailing minus signs to CR to indicate credit entries.

Business editing is controlled by the B edit descriptor which has the form:

B'<string>'

where <string> can contain the following characters:

+ - \$, * Z # . CR

The field width is indicated by the number of characters in <string>. If the field width is too small for the number in question, then the output field will be filled with asterisks.

The characters have the following significance:

PLUS (+)

'FIXED SIGN': if the first character of <string> is a single plus (+), then the actual sign of the number (+ or -) is printed as the first character on the left in the output field.

'FLOATING SIGN': if there are multiple plus (+) signs at the beginning of <string>, then these will be replaced in the output field by printing characters and the actual sign of the number (+ or -) will be printed on the extreme left in the output field.

'TRAILING SIGN': this is the plus (+) sign on the extreme right of <string>. The actual sign (+ or -) of the number will be printed in that output field position.

MINUS (-)

This works in the same way as the PLUS sign. However, for a positive number a blank is printed instead of '+'. This is PLUS sign suppression.

DOLLAR SIGN (\$)

A DOLLAR SIGN sign may not be preceded by anything except a fixed sign. 'FIXED DOLLAR' is a single dollar sign which will be printed in the corresponding position in the output number.

'FLOATING DOLLAR': these are multiple dollar signs which are replaced by printing digits in the output number. A single dollar sign will be printed as the first character on the left.

ASTERISK (*)

If the output number has a digit where there is an asterisk, this digit will be printed. Otherwise, an asterisk (*) will be printed - this is field filling. An asterisk may be preceded only by a fixed sign and/or a fixed dollar.

ZED (Z)

This indicates leading zero suppression. In other words, if the digit in the output number is a leading zero, it will not be printed and a blank space will appear instead.

NUMBER SIGN (#)

Digit positions indicated by #'s are not subject to leading zero suppression.

COMMA (,)

If a comma occurs in the asterisk field, then a “*” will be printed. If a comma is preceded by a significant character (which is not a sign or a dollar sign) then a “,” will be printed in the output field. Otherwise, a blank space will be printed.

Commas must follow any leading characters and precede decimal points.

CREDIT (CR)

The characters CR may only appear as the last two characters of <string>. In the output “CR” will be printed following the number if it is negative, otherwise, two blanks will be printed.

DECIMAL POINT (.)

decimal point in the output number. The only characters allowed to follow the decimal point are #, CR or trailing signs.

The examples in Table 10-4 illustrate the use of the B edit descriptor.

Number	B-Format	Output
147	B'####'	0147
14789	B'####'	****
0	B'####'	0000
147	B'ZZZZ'	147
1478	B'ZZZZ'	1478
0	B'ZZZZ'	
0	B'ZZZ#'	0
6.089	B'#.##'	6.09
0	B'#.##'	0.00
9876.34	B'ZZZ,ZZZ,ZZ#.##'	9,876.34
987654.34	B'ZZZ,ZZZ,ZZ#.##'	987,654.34
0	B'ZZZ,ZZZ,ZZ#.##'	0.00
8	B'+###'	+008
-8	B'+###'	-008
8	B'-ZZ#'	8
-8	B'-ZZ#'	- 8
126	B'ZZZZZ+'	126+
-126	B'ZZZZZ+'	126-
126	B'ZZZZZ- '	126
-126	B'ZZZZZ- '	126-
45678	B'ZZZ,ZZ#CR'	45,678
-45678	B'ZZZ,ZZ#CR'	45,678CR
308	B'+++,++#.##'	+308.00
-308	B'+++,++#.##'	-308.00
99	B'\$ZZZZZ#'	\$ 99
99	B'\$\$\$\$\$\$#'	\$99
308126	B'\$***,***,***#.##'	\$****308,126.00

Table 10-4 Business editing examples

Miscellaneous Input/Output Extensions

- A “\$” or “\” edit descriptor is provided to facilitate the output of requests to the screen for information, without generating a new line. The descriptor must terminate the format specification. The comma preceeding “\” is optional (like it is with the “/” descriptor). For example:

```

      WRITE(*,1)K
1      FORMAT('Old K=',I5,'Enter new value',,$)
      . . .

```

- The use of list-directed input and output with internal files is permitted.
- The use of non-character arrays containing formats is permitted. It is not recommended that this facility be used in new programs.
- The specification of those edit descriptors which involve integer constants has been extended to permit the replacement of any integer by an expression (in diamond brackets) involving integer constants and any **PARAMETER** names.

For example:

```

      PARAMETER (IR=3,IW=9,ID=4)
      . . .
      WRITE(*,10)A,B,C
10      FORMAT(<IR>F<IW>.<ID>)
      . . .

```


Intrinsic functions

Introduction

The ANSI Standard defines a wide variety of functions that operate on data of type `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX` and `CHARACTER`. `FTN77` provides all the intrinsic functions defined in the ANSI Standard together with functions that provide bit-by-bit logical operations, shifts, determination of the storage address of a data item, accessing of integer data directly using its storage address, and operations on the `COMPLEX*16` data type.

Non-ANSI intrinsic functions

All the intrinsic functions provided by `FTN77` can be used without declaration and can be referenced at any point in any program unit, provided that the intrinsic function name has not been used for some other purpose in that program unit. It is, however, recommended that the name of each intrinsic function used in a program unit should appear in an `INTRINSIC` statement for two reasons:

- 1) so that the programmer has a complete record of all intrinsic functions used in a program unit,
- 2) so that a diagnostic should be output if the program is transferred to another computer system whose Fortran compiler does not provide the name of any `FTN77`-specific intrinsic function that has been used.

It is also recommended that the `EXTERNAL` statement is used for all functions that are not intrinsic, for similar reasons.

Two further points should be noted regarding the `FTN77`-specific intrinsic functions:

- 1) If the /ANSI compile-time option is used, FTN77 outputs a warning message if the name of an FTN77 intrinsic function appears in an INTRINSIC statement.
- 2) Other Fortran 77 compilers may provide non-ANSI intrinsic functions whose names are the same as those provided by FTN77 but whose argument types and results may be defined differently.

Generic and specific names

Many intrinsic functions can be referenced in two ways:

- 1) by using a *specific* name whose associated function definition requires a specific type of argument and which will return a result of a specific type,
- 2) by using a *generic* name with a particular type of argument. FTN77 then determines, from the argument type, which equivalent specific function is required. This is possible because the arguments used with any reference to an intrinsic function must be of the same general type, that is, all integer (INTEGER*1, INTEGER*2 or INTEGER*4) or all real etc..

The following pairs of function references have the same effect:

```
IMAX = MAX0(I,J,K)
IMAX = MAX(I,J,K)
```

Here MAX0 is a specific name whilst MAX is a generic name. In order to assist users who wish to remove obsolete specific function names, FTN77 outputs comments such as

```
MAX0 could be replaced by its generic equivalent
(MAX) throughout this program unit
```

If this comment appeared at the end of a program unit, the programmer could safely use the editor to replace the name MAX0 by the name MAX throughout that program unit.

Intrinsic function names as actual arguments

In order to use an intrinsic function name as an argument, it must appear in an INTRINSIC statement in the calling program unit. Only specific function names can

be used in this way. If the generic and specific function names are the same, the specific function is passed as an actual argument. For example

```

      INTRINSIC LOG10, SIN, ALOG10
      REAL LOG10
C   valid use of intrinsic name (SIN is specific):
      CALL FRED(SIN)
C   invalid use of intrinsic name (LOG10 is generic):
      CALL FRED(LOG10)
C   valid use of intrinsic name (ALOG10 is specific):
      CALL FRED(ALOG10)

```

The following ANSI-specific intrinsic function names must not be used as an actual argument:

INT	IFIX	IDINT	FLOAT	SNGL	REAL	CMPLX
LGE	LGT	LLE	LLT	MAX0	AMAX1	DMAX1
AMAX0	MAX1	MIN0	AMIN1	DMIN1	AMIN0	MIN1

In addition, the following FTN77-specific intrinsic function names must not be used as an actual argument:

DFLOAT	LENG	LEQ	LNE	AND	OR	XOR
NOT	INTB	INTS	INTL	CCORE1	CORE1	CORE2
CORE4	FCORE4	DCORE8	LGCB	LGCS	LGCL	RS
LS	RR	LR	RT	LT	SHFT	LOC
BITS	DREAL	DCMPLX	DIMAG			

Integer arguments and function results

The ANSI Standard does not define INTEGER*1, INTEGER*2 and INTEGER*4 data (only type INTEGER) and thus no ANSI intrinsic function definition refers to either of these non-standard data types.

All INTEGER data should be INTEGER*4 data for full ANSI-conformity, and programs should be written using the Fortran keyword INTEGER (not INTEGER*4) and compiled with the /INTL option (this is the default under Win32). However, for reasons of efficiency, storage, history and so on, many programs will contain INTEGER*1, INTEGER*2 data and possibly INTEGER*4 data as well. All the FTN77 intrinsic functions that require integer arguments will accept any combination of INTEGER*1, INTEGER*2 and INTEGER*4 arguments.

The length of the result of an integer intrinsic function is determined as follows:

- For functions that have non-integer arguments (for example, `REAL`) the result is `INTEGER*2` when the compile-time option `/INTS` (the default under DOS/Win16) is used and `INTEGER*4` when the compile-time option `/INTL` is used.
- For functions that have integer arguments (for example, `MAX0`) the result type is `INTEGER*2` unless one or more of the arguments is `INTEGER*4` in which case the result type is `INTEGER*4`.

Full details are given for each function in the table and notes on pages 143 and 148.

Logical arguments and function results

The ANSI Standard does not define `LOGICAL*1`, `LOGICAL*2` and `LOGICAL*4` data (only type `LOGICAL`) and thus no ANSI intrinsic function definition refers to either of these non-standard data types.

All `LOGICAL` data should be `LOGICAL*4` data for full ANSI-conformity, and programs should be written using the Fortran keyword `LOGICAL` (not `LOGICAL*4`) and compiled with the `/LOGL` option (the default under Win32). However, many programs will contain `LOGICAL*1` and `LOGICAL*2` data possibly mixed with `LOGICAL*4` data.

The length of result of a logical function is determined as follows:

- The result is `LOGICAL*2` when the compile-time option `/LOGS` (the default under DOS/Win16) is used and `LOGICAL*4` when the compile-time option `/LOGL` is used.

Full details are given for each function in the table and notes on pages 143 and 148.

Intrinsic function descriptions

The table which follows, used in conjunction with its accompanying notes, give a full description of each intrinsic function provided by FTN77. An FTN77-specific function is indicated by an asterisk following its name in the table.

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Conversion from numeric to integer (1,32,34,35,38)	INT	- INT IFIX IDINT - -	Numeric Integer Real Double Complex Complex*16	Integer Integer Integer Integer Integer Integer	1
Conversion from numeric to byte integer (2,34,36,38)	INTB*	-	Numeric	Integer*1	1
Conversion from numeric to short integer (2,34,36,38)	INTS*	-	Numeric	Integer*2	1
Conversion from numeric to long integer (2,34,36,38)	INTL*	-	Numeric	Integer*4	1
Conversion from numeric to real (3,34,35,38)	REAL	- FLOAT SNGL	Numeric Integer Double	Real Real Real	1
Conversion from numeric to double (4,34,35,38)	DBLE	- DFLOAT* DREAL*	Numeric Integer Complex*16	Double Double Double	1
Conversion from numeric to complex (5,34,35,38)	CMPLX	-	Numeric	Complex	1 or 2
Conversion from numeric to complex*16 (6,34,36,38)	DCMPLX*	-	Numeric	Complex*16	1 or 2

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Conversion from logical to logical*1 (36,43)	LGCB*	-	Logical	Logical*1	1
Conversion from logical to logical*2 (36,43)	LGCS*	-	Logical	Logical*2	1
Conversion from logical to logical*4 (36,43)	LGCL*	-	Logical	Logical*4	1
Conversion from character to integer (7,32,34)	ICHAR	-	Character	Integer	1
Conversion from integer to character (7,34)	CHAR	-	Integer	Character	1
Truncation (39)	AINT	AINT DINT	Real Double	Real Double	1
Nearest whole number (8)	ANINT	ANINT DNINT	Real Double	Real Double	1
Nearest integer (9,32)	NINT	NINT IDNINT	Real Double	Integer Integer	1
Absolute value (10,35,40)	ABS	IABS ABS DABS CABS CDABS*	Integer Real Double Complex Complex*16	Integer Real Double Real Double	1
Modulus (11,33)	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double	2
Transfer of sign (12,33)	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double	2

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Positive difference (33,42)	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double	2
Double precision product	-	DPROD	Real	Double	2
Choosing largest value (33,34)	MAX	MAX0 AMAX1 DMAX1	Integer Real Double	Integer Real Double	≥ 2
(32,34)	-	AMAX0 MAX1	Integer Real	Real integer	≥ 2
Choosing smallest value (33,34)	MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double	≥ 2
(32,34)	-	AMIN0 MIN1	Integer Real	Real Integer	≥ 2
Declared length of character argument (13,32)	-	LEN	Character	Integer	1
Significant length of character argument (36,37)	-	LENG*	Character	Integer	1
Location of substring (argument 2) in string (argument 1) (14,32)	-	INDEX	Character	Integer	2
Real part of Complex argument (15,16,34,35)	-	REAL DREAL*	Complex Complex*16	Real Double	1
Imaginary part of Complex argument (16,34,35)	AIMAG	AIMAG DIMAG*	Complex Complex*16	Real Double	1

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Conjugate of Complex argument (16,35)	CONJG	CONJG DCONJG*	Complex Complex*16	Complex Complex*16	1
Square root (17,35)	SQRT	SQRT DSQRT CSQRT CDSQRT*	Real Double Complex Complex*16	Real Double Complex Complex*16	1
Exponential (35)	EXP	EXP DEXP CEXP CDEXP*	Real Double Complex Complex*16	Real Double Complex Complex*16	1
Natural logarithm (18,35)	LOG	ALOG DLOG CLOG CDLOG*	Real Double Complex Complex*16	Real Double Complex Complex*16	1
Common logarithm (18)	LOG10	ALOG10 DLOG10	Real Double	Real Double	1
Logarithm to base 2 (18,36,41)	LOG2*	ALOG2* DLOG2*	Real Double	Real Double	1 1
Sine (19,21,35)	SIN	SIN DSIN CSIN CDSIN*	Real Double Complex Complex*16	Real Double Complex Complex*16	1
Cosine (19,21,35)	COS	COS DCOS CCOS CDCOS*	Real Double Complex Complex*16	Real Double Complex Complex*16	1
Tangent (19,21)	TAN	TAN DTAN	Real Double	Real Double	1
Arcsine (20,22)	ASIN	ASIN DASIN	Real Double	Real Double	1
Arccosine (20,23)	ACOS	ACOS DACOS	Real Double	Real Double	1
Arctangent arctan(a1) (20,24)	ATAN	ATAN DATAN	Real Double	Real Double	1
Arctan(a1/a2) (20,24)	ATAN2	ATAN2 DATAN2	Real Double	Real Double	2

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Hyperbolic sine (19)	SINH	SINH DSINH	Real Double	Real Double	1
Hyperbolic cosine (19)	COSH	COSH DCOSH	Real Double	Real Double	1
Hyperbolic tangent (19)	TANH	TANH DTANH	Real Double	Real Double	1
Lexically greater than or equal (25,34)	-	LGE	Character	Logical	2
Lexically greater than (25,34)	-	LGT	Character	Logical	2
Lexically less than or equal (25,34)	-	LLE	Character	Logical	2
Lexically less than (25,34)	-	LLT	Character	Logical	2
Lexically equal (25,34,36)	-	LEQ*	Character	Logical	2
Lexically not equal (24,25,34,36)	-	LNE*	Character	Logical	2
Extraction of bit field (34,36,44)	-	BITS*	Integer	Integer	3
Bitwise AND (26,34,36,46)	-	AND* IAND*	Integer	Integer	≥ 2
Bitwise OR (26,34,36,46)	-	OR* IOR*	Integer	Integer	≥ 2
Bitwise XOR (26,34,36,46)	-	XOR* IEOR*	Integer	Integer	≥ 2
Bitwise NOT (27,34,36)	-	NOT*	Integer	Integer	1
Left shift (28,34,36)	-	LS*	Integer	Integer	2

Definition and notes	Generic name	Specific name	Type of arguments	Type of function	No. of arguments
Right shift (28,34,36)	-	RS*	Integer	Integer	2
Left rotate (28,34,36)	-	LR*	Integer	Integer	2
Right rotate (28,34,36)	-	RR*	Integer	Integer	2
Shift (29,34,36)	-	SHFT*	Integer	Integer	2 or 3
Left truncate (30,34,36)	-	LT*	Integer	Integer	2
Right truncate (30,34,36)	-	RT*	Integer	Integer	2
Obtain address (31,36)	-	LOC*	Any	Integer*4	1
Obtain contents of address (34,36,45)	-	CCORE1*	Integer*4	Character*	1
		CORE1*	Integer*4	Integer*1	1
		CORE2*	Integer*4	Integer*2	1
		CORE4*	Integer*4	Integer*4	1
		FCORE4*	Integer*4	Real	1
		DCORE8*	Integer*4	Double	1

Notes for the table of intrinsic functions

In the following notes the names of data types are given in lowercase; uppercase is reserved for intrinsic function names.

- 1) The generic INT discards the fractional part of its argument, producing a truncated (unrounded) integral value. The result will be INTEGER*2 in a program unit compiled with /INTS, and INTEGER*4 in a program unit compiled with /INTL.
- 2) INTB, INTS and INTL are similar to INT, differing only in that the result-type is determined by the function selected rather than the compiler option in effect.
- 3) For x of type real, REAL(x) is x . For x of type integer or double precision, REAL(x) is as much precision of x as a real datum, can contain. For x of type complex, REAL(x) is the real part of x .
- 4) For x of type double precision, DBLE(x) is x . For x of type integer or real, DBLE(x) is the value of x in double precision form. For x of type complex,

$\text{DBLE}(x)$ is the real part of x in double precision form.

- 5) CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision.

For x of type complex, $\text{CMPLX}(x)$ is x . For x of type integer, real, or double precision, $\text{CMPLX}(x)$ is the complex value whose real part is $\text{REAL}(x)$ and whose imaginary part is zero. $\text{CMPLX}(x1, x2)$ is the complex value whose real part is $\text{REAL}(x1)$ and whose imaginary part is $\text{REAL}(x2)$.

- 6) DCMPLX is similar to CMPLX , except that a COMPLEX^*16 number is produced.
- 7) Every character is represented by FTN77 as a sequence of eight bits ranging from 00000000 - 11111111 (decimal 0 to 255). Any such sequence can be interpreted either as a character or as an integer. CHAR and ICHAR provide a means for converting between the two interpretations.

ICHAR operates on a single character. It returns an integer between 0 and 255, representing the decimal equivalent of the bit pattern for that character.

CHAR operates on any integer. If the integer is between 0 and 255, it is used directly. Otherwise, it is converted to the range 0 to 255 by truncating all but the eight rightmost bits (the lowest order byte).

Following conversion, if required, CHAR returns the character whose bit pattern corresponds to the binary equivalent of its argument.

The ASCII character set is used by FTN77 for formatted CHARACTER input/output operations and for CHARACTER constants.

- 8) $\text{ANINT}(x)$ is defined as:

$$\begin{aligned} &\text{REAL}(\text{INTL}(x+.5)) \quad \text{if } x \geq 0 \\ &\text{REAL}(\text{INTL}(x-.5)) \quad \text{if } x < 0 \end{aligned}$$

$\text{DNINT}(x)$ is defined as:

$$\begin{aligned} &\text{DBLE}(\text{INTL}(x+.5)) \quad \text{if } x \geq 0 \\ &\text{DBLE}(\text{INTL}(x-.5)) \quad \text{if } x < 0 \end{aligned}$$

- 9) $\text{NINT}(x)$ and $\text{IDNINT}(x)$ are defined as:

$$\begin{aligned} &\text{INT}(x + .5) \quad \text{if } x \geq 0 \\ &\text{INT}(x - .5) \quad \text{if } x < 0 \end{aligned}$$

- 10) The argument to IABS may be INTEGER^*1 , INTEGER^*2 or INTEGER^*4 . The result will be of the same type as the argument.

- 11) **MOD** yields the remainder when its first argument is divided by its second argument. Both arguments must be of the same type; the result will also be of that type.

The three variants of **MOD** are defined as follows:

$$\begin{aligned}\text{MOD}(x1,x2) &= x1 - (\text{INTL}(x1/x2) * x2) \\ \text{MOD}(x1,x2) &= \text{REAL}(x1 - (\text{INTL}(x1/x2) * x2)) \\ \text{MOD}(x1,x2) &= \text{DBLE}(x1 - (\text{INTL}(x1/x2) * x2))\end{aligned}$$

The result for **MOD**, **AMOD**, and **DMOD** is a “Division by Zero” error when the value of the second argument is zero.

- 12) This function combines the magnitude of its first argument with the sign of the second. If the value of the first argument is zero, the result is zero, which is neither positive nor negative.

The variants of **SIGN** produce the following result:

$$|x1| \text{ if } x2 \geq 0 ; -|x1| \text{ if } x2 < 0$$

where $x1$ and $x2$ are the two arguments.

- 13) The value of the argument of the function **LEN** need not be defined at the time the function reference is executed.

- 14) **INDEX**($x1,x2$) returns an integer value indicating the starting position within the character string $x1$ of a substring identical to string $x2$. If $x2$ occurs more than once in $x1$, the starting position of the first occurrence is returned.

If $x2$ does not occur in $x1$, the value zero is returned. Note that zero is returned if $\text{LEN}(x1) < \text{LEN}(x2)$.

- 15) The **REAL** function for real-part extraction is the same specific function that is selected when the generic function **REAL** is given a **COMPLEX*8** argument.

The **DREAL** function for real-part extraction is the same specific function that is selected when the generic function **DBLE** is given a **COMPLEX*16** argument.

REAL and **DREAL** for real-part extraction cannot be passed as arguments in Fortran 77 because they are specific type-conversion functions. To provide symmetry with **AIMAG** and **DIMAG** imaginary-part extraction, which can be passed, FTN77 allows **REAL** and **DREAL** passed as arguments.

- 16) A complex value is expressed as an ordered pair of reals, (xr,xi), where xr is the real part and xi is the imaginary part.

- 17) The value of the argument of **SQRT** and **DSQRT** must be greater than or equal to zero. The result of **CSQRT** and **CDSQRT** is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

- 18) The value of the argument of ALOG, DLOG, ALOG10, DLOG10 and DLOG2 must be greater than zero. The value of the argument of CLOG and CDLOG must not be (0.,0.). The result of CLOG and CDLOG is the principal value, i.e. the range of the imaginary part of the result is

$$-\pi < \text{imaginary part} \leq \pi$$

The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

- 19) All angles are expressed in radians.
 20) The result will be expressed in radians.
 21) The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN is not restricted to be less than 2π .
 22) The absolute value of the argument of ASIN and DASIN must be less than or equal to 1. The range of the result is:

$$-\pi/2 \leq \text{result} \leq \pi/2$$

- 23) The absolute value of the argument of ACOS and DACOS must be less than or equal to 1. The range of the result is:

$$0 \leq \text{result} \leq \pi$$

- 24) The range of the result for ATAN and DTAN is:

$$-\pi/2 \leq \text{result} \leq \pi/2$$

If the value of the first argument of ATAN2 or DTAN2, (the y coordinate in the cartesian x,y pair) is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$.

The arguments must not both have the value zero.

The range of the result for ATAN2 and DATAN2 is:

$$-\pi < \text{result} < \pi$$

- 25) LGE(*x1*,*x2*) returns the value .TRUE. if *x1* = *x2* or if *x1* follows *x2* in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value .FALSE.

LGT(*x1*,*x2*) returns the value .TRUE. if *x1* follows *x2* in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value .FALSE.

`LLE(x1,x2)` returns the value `.TRUE.` if $x1 = x2$ or if $x1$ precedes $x2$ in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value `.FALSE.`

`LLT(x1,x2)` returns the value `.TRUE.` if $x1$ precedes $x2$ in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value `.FALSE.`

`LEQ(x1,x2)` returns the value `.TRUE.` if $x1 = x2$ and otherwise returns the value `.FALSE.` `LEQ` is an FTN77-specific intrinsic function.

`LNE(x1,x2)` returns the value `.TRUE.` if $x1$ is not equal to $x2$ and otherwise returns the value `.FALSE.` `LNE` is an FTN77-specific intrinsic function.

If the operands for `LGE`, `LGT`, `LLE`, `LLT`, `LEQ`, and `LNE` are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

The result-type for `LGE`, `LGT`, `LLE`, and `LLT` will be `LOGICAL*4` in a program unit compiled with `/LOGL`, and `LOGICAL*2` in a program unit compiled with `/LOGS`.

As FTN77 uses the ASCII collating sequence, the use of these functions produces exactly the same result as a comparison of $x1$ and $x2$. Thus, for example,

`LLE(C1,C2)` and `C1.LE.C2`

are equivalent but the same may not be true for other Fortran implementations. Thus `LLE`, `LLT`, `LGE` and `LGT` should always be used in portable programs where the collating sequence used for comparisons must be known exactly.

- 26) `AND`, `OR`, and `XOR` perform the bitwise logical function named on a list of long, short and byte integers. The result will be a long integer if any argument is long; otherwise it will be a short integer. When byte, short and long integers are mixed, the byte and short integers will be sign-extended, *not* zero-extended.
- 27) Performs a bitwise logical `NOT` function (ones complement) on a long, short or byte integer. The result has the type of the argument.
- 28) `LS`, `RS`, `LR` and `RR` take two arguments; each argument may be either a long or a short integer. These arguments are called `ARG1` and `ARG2` in the following.

`LS` shifts `ARG1` to the left by the number of bits specified in `ARG2`. The result has the type of `ARG1`, that is, no type change occurs. Vacated places are filled with zeros. If `ARG2` is zero, no shift occurs. If `ARG2` is negative, the effects of the operation are undefined.

`RS` is identical to `LS`, except that the shift is to the right.

`LR` rotates the bits in `ARG1` by the number of bits specified in `ARG2`. The result has the type of `ARG1`, that is, no type change occurs. Bits are removed from the left hand end of `ARG1` and replaced at the right hand end.

RR is similar to LR except that the rotation takes place in the opposite direction; bits from the right hand end are replaced at the left hand end.

- 29) SHFT is similar to LS and RS, except that it can shift in either direction, and can perform two shifts rather than one. The additional shift occurs if a third integer argument, ARG3, is given.

If ARG2 is negative, the shift is to the left; if it is positive, the shift is to the right; if it is zero, no shift occurs.

If ARG3 appears, the shift specified by it will be carried out after the shift specified by ARG2 is complete. The rules are the same as for the ARG2 shift.

Note that the sense of the shift specified by a positive or negative value of ARG2 or ARG3 is not the same as that defined for the equivalent functions provided by IBM and DEC Fortran (77) compilers.

- 30) LT takes two arguments; each argument may be either a long or a short integer. These arguments are called ARG1 and ARG2 in the following.

LT preserves the left ARG2 bits of ARG1, and sets the rest to zero (left truncation). The result has the type of ARG1 - that is, no type-change occurs. If $\text{ARG2} \leq 0$, no bits are preserved.

RT is identical to LT, except that the right ARG2 bits are preserved.

- 31) LOC operates on an item of any data type. The result is an INTEGER*4 value representing the memory address where the first byte of the data item is located.
- 32) An integer result produced by this function will be INTEGER*2 in a program unit compiled with /INTS, and INTEGER*4 in a program unit compiled with /INTL.
- 33) When this function operates on integers, the arguments may be a mixture of INTEGER*1, INTEGER*2 and INTEGER*4. The result will have the type of the longest argument.

A special case arises when IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram. In this case, the invoking program unit has no opportunity to examine the argument list on which the function will operate. Therefore it cannot select the version of the function that will implement the above rule. For compatibility with the ANSI Standard, the following rule is used instead:

When IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram, the function passed will accept and produce INTEGER*4 values if the invoking program unit was compiled with /INTL, and INTEGER*2 values if it was compiled with /INTS. This is the only case in which integer types cannot be mixed in the argument list of an integer intrinsic function.

- 34) This function cannot be passed as an argument to a subprogram.

- 35) The specific function accepting the COMPLEX*16 data type is an FTN77 extension.
- 36) This function is an FTN77 extension.
- 37) LENG(*x*) returns an integer in the range 0 to LEN(*x*). It is the length of the character argument *x* after any trailing blanks have been removed.
- 38) "Numeric" means any of REAL, INTEGER*1, INTEGER*2, INTEGER*4, DOUBLE PRECISION, COMPLEX or COMPLEX*16.
- 39) AINT(*x*) is equivalent to REAL(INTL(*x*)). DINT(*x*) is equivalent to DBLE(INTL(*x*)).
- 40) CABS and CDABS are defined as follows:

$$\sqrt{xr^2 + xi^2}$$

where *xr* and *xi* are, respectively, the real and imaginary parts of the complex number.

- 41) DLOG2 is an FTN77-specific intrinsic function.
- 42) The variants of DIM produce the following result:

$$\begin{array}{lll} x1 - x2 & \text{if} & x1 > x2 \\ 0 & \text{if} & x1 \leq x2 \end{array}$$

where *x1* and *x2* are the two arguments.

- 43) The result of LGCB(*x*) is LOGICAL*1, the result of LGCS(*x*) is LOGICAL*2 and the result of LGCL(*x*) is LOGICAL*4. These functions are provided so that logical subroutine and function arguments can be converted to the correct length (cf. INTB, INTS and INTL).
- 44) BITS(*i,n,m*) extracts a bit field from integer *i* which can be either INTEGER*2 or INTEGER*4. The result of the function is an integer of the same length as *i* that contains, as its least significant bits, bits *n* to *m* inclusive of *i*. The remaining bit positions of the result are set to 0.
- n* and *m* should be in the range 1 to 8, 1 to 16 or 1 to 32, respectively, for *i* of type INTEGER*1, INTEGER*2 or INTEGER*4. *n* should not be greater than *m*. Values of *n* and *m* that do not conform to these rules will produce undefined results. For example:

```
INTEGER*2 I2,A2
INTEGER*4 I4,A4
A2 = :35          /* same as 0'35'
I2 = BITS(A2,14,15)
```

```
A4 = :100          /* same as 0'100'
I4 = BITS(A4,26,26)
```

would result in I2 and I4 having the values 2 and 1 respectively. Wherever possible, BITS is implemented by means of in-line code.

- 45) CCORE1, CORE1, CORE2, CORE4, FCORE4 and DCORE8 are used to manipulate data using addresses that are known to the programmer (for example, by using LOC). Each function takes one INTEGER*4 argument that is an address.

The functions can appear on the right hand side of an assignment statement if they are declared in an INTRINSIC statement. For example:

```
INTRINSIC CORE2
INTEGER*4 I4
I4 = LOC(J)
CORE2(I4) = CORE2(I4) + 1
```

The second assignment statement is equivalent to:

```
J = J + 1
```

This trivial example illustrates how CORE1, CORE2 and CORE4 might be used in software that involves data structures containing addresses.

- 46) An alternative name, which will have the correct implied type, is listed for this function.

Fortran 77 character handling facilities

The character handling facilities of Fortran 77 are a major feature of the language. It is possible to manipulate data of type **CHARACTER** in a way which enables truly portable programs to be written. Type **CHARACTER** is not the same as Hollerith data but is intended to replace the many forms of Hollerith data manipulation that are available in the various implementations of Fortran 66.

Character statements

The **CHARACTER** statement is used in the same way as any other type statement. For example:

```
CHARACTER A,B,C
```

declares variables **A**, **B** and **C** to be of type **CHARACTER**. Every variable of type **CHARACTER** has associated with it a length (the variables in the above example each have length 1) which must be specified explicitly or implicitly in the **CHARACTER** statement. In general a **CHARACTER** statement consists of

- The keyword **CHARACTER**
- An optional length specification which takes the form **length*
- A comma which is optional and therefore usually omitted
- A list of names and array declarators. A name may be that of a variable, an array, a **PARAMETER** (symbolic name of a constant) or a function. Each name or array declarator may be optionally followed by **length*.

For example:

```
CHARACTER*10 W,X,Y*3,Z(10)*4
```

declares W, X, Y and Z to be of type CHARACTER. The length specification *10 is applied to each name in the CHARACTER statement in the absence of further specifications. Thus

```
W    is of length 10
X    is of length 10
Y    is of length 3 (*3 overrides the statement default of *10)
Z    each element of the array Z is of length 4
```

It is not possible to tell merely from the type statements whether W, X and Y are the names of variables, arrays or functions (the same is true of the INTEGER, REAL etc. type statements). In general, **length* in a CHARACTER statement may be one of the following:

- Omitted, for example:

```
CHARACTER A,B,C*3
```

If **length* does not follow the keyword CHARACTER a length of 1 is assumed for each name in the list which does not have its own specification. Conversely, if **length* does not follow a particular name, the statement default is used.

- An unsigned, non-zero integer constant (as in the examples above).
- An integer constant expression which has a non-zero positive value. The expression must be enclosed in parentheses, for example:

```
INTEGER L1,L2
PARAMETER (L1=6, L2=9)
CHARACTER*(L1+L2) C1*(L2), C2*(L1), C3
```

- declares C1 to be of length 9, C2 to be of length 6 and C3 to be of length 15.
- An asterisk enclosed in parentheses (*). This length specification can only be used with a name of one of the following types:
 - **Parameter name:** In this case, the actual length of the name is determined by the length of the defining character constant expression.
 - **Dummy argument name:** In this case, the length assumed by the dummy argument is that of the associated actual argument used whenever the subroutine or function is invoked.
 - **Function subprogram name:** In this case, the length assumed by the name is that which has been specified in the calling program.

Character constants

A character constant is written as a non-empty string of characters enclosed in apostrophes. For example, 'ABCDE' is a character constant of length 5. The initial and final apostrophes do not form part of the constant and are not stored. Space characters occurring in a character constant are significant so that, for example:

```
'BEAR MOUNTAIN'
```

is a character constant of length 13 but

```
'BEARMOUNTAIN'
```

is a character constant of length 12. The two constants clearly have different values. If an apostrophe is required as part of a character constant, each significant apostrophe should be represented by two consecutive apostrophes in the source program. For example:

```
'THE LION''S DEN'
```

is a character constant of length 14. The value of this constant is THE LION'S DEN which shows that the way a constant is written and its value are not necessarily the same thing. A final example shows the rather cumbersome notation required when the value includes surrounding apostrophes:

```
'''THE MOUSETRAP'''
```

The value of this constant is 'THE MOUSETRAP' - its length is 15.

Character expressions

A character expression is one of the following:

- 1) A character constant
- 2) The symbolic name of a character constant (PARAMETER)
- 3) A character variable name
- 4) A character array element reference
- 5) A character substring
- 6) A character function reference
- 7) An expression formed by combining two or more of items 1 to 6 by means of the *concatenation operator* // as follows:

<first string> // <second string>

The value of this expression is <first string> concatenated on the right with <second string>. For example:

Expression	Value
'ALPHA' //'BET'	ALPHABET
'PEACHES' //' AND ' //'CREAM'	PEACHES AND CREAM

The second example above shows that any number of concatenations may appear in an expression. Note that any combination of items 1 to 7 can be used to form an expression and that brackets may be used freely so that the following are exactly equivalent to the second example given above:

```
( 'PEACHES' //' AND ' ) //'CREAM'  
'PEACHES' //' ( ' AND ' //'CREAM'
```

The brackets never have any effect.

Notes:

- ☐ The length of the expression resulting from a concatenation is the sum of lengths of the character operands.
- ☐ Trailing spaces are not removed by concatenations.

Character expressions can appear in character assignment statements, relational expressions and as actual arguments to subroutine and function calls.

Character assignments

A character assignment has the following form:

ch = *character expression*

where *ch* can be a character variable name, a character array element or a character substring. The effect of this statement is to assign the value of *character expression* to *ch*. If the length of *character expression* is less than the length of *ch*, the result in *ch* is padded to the right with spaces. If the length of *character expression* is greater than the length of *ch*, the result is truncated on the right. The following Fortran 77 fragment illustrates some simple character assignments:

```
CHARACTER REGN0*8,MAKE*10,MODEL*10,OWNER*20,  
+ CAR*25,DETAILS*30,INDEX*3
```

C

```

C      assignment of a constant value:
C
      MAKE = 'FORD'
      MODEL = 'CORTINA'
      REGNO = 'XYZ 123W'
      OWNER = 'JACK THE RIPPER'
C
C      concatenation and assignment
C
      CAR = MAKE//MODEL
      DETAILS = REGNO//' '//OWNER
      INDEX = REGNO

```

After these assignment statements have been executed, CAR, DETAILS and INDEX would have, respectively, the following values:

```

FORD▽▽▽▽▽▽▽▽CORTINA▽▽▽▽▽▽▽▽
XYZ▽123W▽JACK▽THE▽RIPPER▽▽▽▽▽
XYZ

```

The character '▽' here, and in the examples which follow, denotes a space. Notice that padding spaces have been used in the assignment to CAR and that the assignment to INDEX has resulted in truncation.

Character expressions in parameter statements

The **PARAMETER** statement can be used to define a symbolic name for a constant of type character. Any character constant expression (that is an expression involving only character constants and symbolic names for character constants) may appear in such a **PARAMETER** statement. For example:

```

      CHARACTER*5 NAME1, NAME2, OPTION*(*)
      PARAMETER (NAME1='SMITH',NAME2='JONES')
      PARAMETER (OPTION=NAME1//' OR '//NAME2)

```

After these **PARAMETER** statements have been processed, NAME1, OPTION, and NAME2 would be symbolic names for the following constants respectively:

```

      SMITH
      JONES
      SMITH OR JONES

```

Note that the length of `OPTION` is now 14 as a result of the `PARAMETER` statement. These symbolic names could be used anywhere in the remainder of the program unit to represent the values listed above so, for example:

```
X = 'SMITH'  
X = NAME1
```

are identical assignment statements.

Character arrays

The reader is assumed to be familiar with the concept of an array. An array name is just a collective name for a number of related items of storage, each of which holds a value of the same type. A character array follows the general rules for arrays of other types. A character array in Fortran 77 can have up to 7 dimensions. Character arrays can either be defined by means of a `CHARACTER` statement or by means of `CHARACTER`, `DIMENSION` and `COMMON` statements, for example:

```
CHARACTER A(10)*5, B(20,20)  
CHARACTER*30 NAMES  
DIMENSION NAMES(100)
```

The above example defines `A` to be an array of 10 elements each of 5 characters. `B` is an array of 400 elements (20 x 20) each of 1 character. `NAMES` is a 100 element array, each element being 30 characters in length.

Character substrings

It is often necessary to use only part of a character variable or array element. Consider the following:

```
CHARACTER*6 FLTNO  
FLTNO = 'BA 748'
```

Suppose that the first two characters of `FLTNO` are needed for some reason. There is a need for a notation that refers to these two characters and yet treats them as a single entity. It might be possible to redefine `FLTNO` as a 6-element array of 1-character elements.

Thus:

```
CHARACTER FLTNO(6)
```

but this raises the problem of assigning the value BA 748. Returning to the previous definition of `FLTNO`, substring notation can be used to extract the desired characters.

```
CHARACTER FLTNO*6,AIRLN*2,NUMBER*3
FLTNO = 'BA 748'
AIRLN = FLTNO(1:2)
NUMBER = FLTNO(4:6)
```

`FLTNO(1:2)` is referred to as a substring name consisting of characters 1 and 2 of `FLTNO`. `FLTNO(4:6)` consists of characters 4 to 6 of `FLTNO`. A substring name can also be formed using an array element name as in the following example, in which the previous simple character variables have been redefined as character arrays:

```
FLTNO(10) = 'BA 748'
AIRLN(10) = FLTNO(10)(1:2)
NUMBER(10) = FLTNO(10)(4:6)
```

A substring name can be formed from an array element of any number of dimensions, for example, `TABLE(5,6)(10:20)`

Data statements involving character entities

A `DATA` statement can be used to initialise a character variable, character array element or character substring. The program fragment below gives some examples:

```
CHARACTER A*6,B*3,C*8,D(10)*4,E*20
C  initialisation of simple variables
  DATA A/'ABCDEF'/ B /'WXYZ'/ C /'PQR'/
C  use of implied DO-loop
  DATA (D(J),I=1,6)/3*'XXXX',2*'YYYY','ZZZZ'/
C  substring initialisation
  DATA E(10:20)/'SECOND HALF'/
```

When the above program is loaded, the variables `A`, `B` and `C` would have the values `ABCDEF`, `WXY` and `PQRVVVVV` respectively. Note that if the length of the variable and defining constant are not the same, then padding or truncation takes place as in the case of assignment statements. The first 6 elements of the array `D` would have the values `XXXX`, `XXXX`, `XXXX`, `YYYY`, `YYYY` and `ZZZZ` respectively but the remaining four elements would be undefined. Characters 10 to 20 of variable `E` would be defined with the value `SECOND HALF` but characters 1 to 9 would be undefined.

Note:

Fortran 77 does not allow an implied `DO-loop` variable to be used to initialise a

character substring. The character substring expressions must always be constant expressions in a **DATA** statement.

Input and output of character data

List-directed input/output is the simplest way to read and print character data. It is only necessary to use an appropriate character entity in the input/output list of a **READ** or **PRINT** statement for its value to be transmitted. For example:

```
CHARACTER C*10, DAY(7)*9
DATA C/' TODAY IS '/
DATA DAY /'MONDAY','TUESDAY','WEDNESDAY',
+ 'THURSDAY','FRIDAY','SATURDAY','SUNDAY'/
1 READ *,NDAY
  IF (NDAY.LT.1.OR.NDAY.GT.7) THEN
    PRINT *, 'ERROR IN NDAY VALUE'
    GOTO 1
  ENDIF
  PRINT *,C, DAY(NDAY)
  . . .
```

In the above example, the first **PRINT** statement outputs the character constant **ERROR IN NDAY VALUE**. The input/output list of the second **PRINT** statement consists of two character entities: a character variable and a character array element.

The rules for list-directed input of character information are slightly more complicated. Consider the statements:

```
CHARACTER*10 C
READ *,C
```

The character data item on the input record corresponding to **C** must have the form of a character constant, that is, the actual value must be enclosed in apostrophes. Spaces are significant and, if an apostrophe character is required as part of the value, it must be represented by two apostrophes on the data record. For example, the values **DOG** and **DOG'S** would be supplied as data, respectively, as

```
'DOG'
'DOG''S'
```

If either of these values were read by the above **READ** statement, the resulting value held in **C** would be padded on the right with spaces as the length (10) of **C** is greater than the length of the constant in each of the above cases. In general, the rules are

exactly those for character assignment. Suppose that the following constants were supplied as an item of data for the above **READ** statement:

```
'0123456789ABCDE'
```

then the resulting value assigned to **C** would be 0123456789. The leftmost characters of the character constant are retained if the length of **C** is less than the length of the data item. Character variables can, of course, appear in input/output lists together with variables of other types. Character values on data records can be repeated like any other value so, for example:

```
CHARACTER C(10)*5
INTEGER N
READ *,N,(C(I),I=1,N)
```

could be used to read the following record

```
6,6*'EMPTY'
```

Six elements of the array **C** would be initialised to the value **EMPTY** by means of this statement.

Note:

The **FTN77** compiler will not accept repeated character constants split over more than one record. This violates the Fortran 77 standard, but is not a serious restriction in practice.

Formatted input/output of character data requires the use of the **Aw** editing descriptor. In general, if the length of the character item in the input/output list is c , there are a number of differing effects for input and output depending on the relative values of w and c . These effects are summarised by the table below.

	Input	Output
$w > c$	The rightmost c characters of the specified field are transferred to the list item	The value of the output list item is transferred to the output field preceded by $w - c$ space characters
$w < c$	The w characters of the specified field are transferred to the list item padded on the right by $c - w$ spaces	The leftmost w characters of the value of the list item are transferred to the output field
w omitted	A field width of c is assumed so that c characters are transferred to/from the specified field	

The examples below illustrate the use of this descriptor and some of the problems that it can cause.

```
CHARACTER C*10  
READ (1,'(A10)')C
```

Ten characters would be transferred from the input record to variable **C**.

```
CHARACTER CA(10)  
READ (1,'(A10)')CA
```

Ten records would be read; the first character of each of these records would be used as a value for the elements. 1 to 10 of array **CA**.

```
CHARACTER SMALL*5,BIG*10  
READ (1,20) SMALL  
READ (1,20) BIG  
20  FORMAT (A80)
```

Suppose the data records corresponding to the above **READ** statements were

```
ALPHABET  
CABBAGES
```

after the **READ** statements had been executed, **SMALL** would have the value **HABET** and **BIG** would have the value **CABBAGESVV**. Note that the effect obtained for **SMALL** is exactly the opposite to that for list-directed input and is, indeed, the reverse of one's expectation.

```
CHARACTER SMALL*5,BIG*10,OK*8  
READ (1,20)SMALL  
READ (1,20)BIG  
READ (1,20)OK  
20  FORMAT (A)
```

In the above example, no field width is specified following **A** in the **FORMAT** statement. In this case, the width assumed is the length of the corresponding character variable or array element in the input/output list. Thus, with data records

```
ALPHABET  
CABBAGES  
KINGS
```

the values obtained in **SMALL**, **BIG** and **OK** after execution of the above **READ** statements would be, respectively, **ALPHA**, **CABBAGESVV** and **KINGSVVV** as might be expected intuitively. The following examples assume unit 2 is connected to a file which does not require carriage control characters.

```
CHARACTER BUFOUT*80  
WRITE (2,'(A80)')BUFOUT
```

80 characters would be transferred from the character variable **A** to the output record.


```

      CHARACTER LITTLE*10,LARGE*26
      LITTLE = '0123456789'
      LARGE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      WRITE (2,10)LITTLE
      WRITE (2,10)LARGE
10    FORMAT (A15)

```

The output records produced by the two **WRITE** statements would be

```

      0123456789
      ABCDEFGHIJKLMNO

```

If the **FORMAT** statement in the above example were replaced by

```

10    FORMAT (A)

```

the output records would be

```

      0123456789
      ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

Comparing character strings

There are a number of applications where character strings must be compared, for example:

```

      CHARACTER*5 CONTRL(3),KEY
      DATA CONTRL/'START','STOP','WAIT'/
6     READ (1,'(A)')KEY
      DO 4 I=1,3
        IF (KEY.EQ.CONTRL(I))GOTO(1,2,3),I
4     CONTINUE
      PRINT *,'invalid control card'
      GOTO 6
1     . . .
2     . . .
3     . . .

```

In this example, the actual internal codes used by the system for the character values do not matter as the test is only for equality of two character strings. Many programs involve the sorting of data using some form of key which is frequently alphanumeric but need not necessarily be so. Sorting implies comparisons of the form 'is the value of A less than the value of B?' or 'is the value of C greater than the value of D?'. This type of comparison implies a need for some rules for ordering character values.

Everyone would intuitively place the words DOG, CAT, RABBIT and MONKEY in the following order:

```
CAT
DOG
MONKEY
RABBIT
```

Suppose that the phrase RABBIT'S FOOT is to be inserted into the above sequence. Clearly, it would follow MONKEY but should it follow RABBIT? Most people would probably answer 'yes' to this question but what would happen if this operation were part of a Fortran 77 program? The answer is that the ordering would depend on the exact details of the collating sequence used by the implementor. FTN77 uses the ASCII collating sequence but the Fortran 77 Standard only defines the following collating sequence.

- 1) $0 < 1 < 2 \dots < 9$
- 2) $A < B < C \dots < Z$
- 3) $9 < A$ or $Z < 0$ (in other words, all the digits must either precede A or follow Z)
- 4) Space < 0 and space $< A$

So, whilst many Fortran 77 implementations do use the ASCII collating sequence, the above minimal rules can lead to non-portable programs unless one of the intrinsic functions LLT, LLE, LGE or LGT is used.

In general, when comparing the values of two character expressions *c1* and *c2* in a relational expression, the shorter expression is assumed to be padded on the right with spaces. A character relational expression has the general form:

$$c1 .op. c2$$

where *c1* and *c2* are character expressions and *.op.* is one of .LT. .LE. .EQ. .GE. .GT. or .NE.

Note:

The effect of .EQ. and .NE. is independent of the collating sequence used.

The example below shows some relational expressions:

```
CHARACTER C*(*),C1*5,C2*4,C3*9
LOGICAL L
PARAMETER (C='APPLE')
PARAMETER (L=C.GT.'A')

      . . .
      C3 = C1//C2
C  does C3 start with an alphabetical character?
      IF (C3.GE.'A') THEN
```

```

        IF (C3.LE.'Z') GOTO 10
    ENDIF
C  no - is it numeric?
        IF (C3.GE.'0') THEN
            IF (C3.LE.'9') GOTO 20
        ENDIF
C  is it blank?
        IF (C3.EQ.' ') GOTO 30
        . . .

```

Intrinsic functions for handling character data

There are eight very useful Fortran 77 intrinsic functions provided specifically to simplify the manipulation of character data. The functions are ICHAR, CHAR, LEN, INDEX, LLE, LGE, LGT and LLT.

Conversion from character to integer and vice-versa

ICHAR(*c*) returns the position of character *c* in the collating sequence used by the system.

CHAR(*i*) returns the *i*th character in the collating sequence used by the system. Note that *i* starts at 0, not 1, and that a single character is returned. The maximum value for *i* when using FTN77 is 255.

FTN77 uses the ASCII collating sequence but other Fortran 77 implementations may not, so use of these functions does not necessarily lead to portable programs.

Example:

```

    CHARACTER VALUE
    READ *,I
    IF(I.LT.0.OR.I.GT.9) THEN
        PRINT *, 'ERROR'
        STOP 999
    ELSE
        VALUE = CHAR(I)
    ENDIF

```

This example converts the value of the single digit integer to character form. Note that:

```

    I = 46
    VALUE = CHAR(I)

```

would not produce the character value 46 as **CHAR** only returns a single character.

```
INTEGER ASTART, ZERO, SPACE
ASTART = ICHAR('A')
ZERO   = ICHAR('0')
SPACE  = ICHAR(' ')
```

This example finds the position of the character **A**, zero and space in the collating sequence.

Length of a character entity

It is often useful to know the length of a character entity, particularly in a subroutine or function where an argument of type character can have an assumed length passed with it. In general, **LEN(c)** returns the length of the character expression *c*. For example:

```
CHARACTER A*10, B(5)*4, C*20
I = LEN(A)
J = LEN(B(1))
K = LEN(C(11:20))
```

After these statements had been executed, **I**, **J** and **K** would have the values 10, 4 and 10 respectively. Note that the length returned by **LEN** is the declared or assumed length of the character expression and not the length of the expression once trailing spaces have been removed. (The **FTN77** compiler provides the function **LENG** for this latter purpose: see chapter 11.)

Locating a substring

The function **INDEX(C, CSUB)** returns the starting position (≥ 1) of the substring **CSUB** in the string **C**.

If **CSUB** is not a part of **C**, **INDEX** returns the value 0. If there is more than one occurrence of **CSUB** in **C**, the starting position of the first occurrence is returned. For example, in order to read a card and establish whether the keyword **SUBROUTINE** appears in column 7 or later, the following program fragment might be used:

```
INTEGER POS
CHARACTER CARD*80, NAME*6
1  READ (1,10) CARD
10 FORMAT (A)
   POS = INDEX (CARD,'SUBROUTINE')
C  subroutine found
C  ensure only blank characters precede it
   IF (CARD(1:POS-1) .NE. ' ') THEN
       PRINT *, 'invalid card'
```

```

      ELSE
C   look for name of subroutine
C   LEN ('SUBROUTINE') is converted to a constant at
C   compile-time in the FTN77 implementation
C   so there is no loss of efficiency here
        DO 30 I=POS+LEN('SUBROUTINE'),80
            IF(CARD (I:I) .NE. ' ') GOTO 20
30      CONTINUE
C   assume name has no embedded spaces
20      NAME = CARD(I:I+5)
      ENDIF
      . . .

```

Portable character comparisons

The results of character comparisons which use `.EQ.` and `.NE.` are independent of the collating sequence used but the results of character comparisons using `.LT.`, `.LE.`, `.GE.` and `.GT.` depend on the collating sequence used. To overcome this problem, Fortran 77 provides four logical intrinsic functions which use the ASCII character set for character comparisons:

Function	Equivalent to
LLT (<i>c1</i> , <i>c2</i>)	<i>c1</i> .LT. <i>c2</i>
LLE (<i>c1</i> , <i>c2</i>)	<i>c1</i> .LE. <i>c2</i>
LGE (<i>c1</i> , <i>c2</i>)	<i>c1</i> .GE. <i>c2</i>
LGT (<i>c1</i> , <i>c2</i>)	<i>c1</i> .GT. <i>c2</i>

As FTN77 uses the ASCII collating sequence, there is no need to use the above functions unless a portable program is being written. For example, the value of the relational expression

```
'KEY1A' .GT. 'KEYA1'
```

depends on the relative position of 1 and A in the collating sequence.

```
LGT('KEY1A', 'KEYA1')
```

will always return the value `.FALSE.` as the character 1 precedes the character A in the ASCII collating sequence.

Character functions

The FTN77 library contains some functions of type character, one of which is `DATE()`. This function must be declared in the calling program unit as

```
CHARACTER DATE*8
```

It returns today's date as a character value in the form mm/dd/yy. The following statements could be used to print the date:

```
CHARACTER DATE*8
PRINT *,DATE()
```

This character function has no arguments. In general, it is possible to write a character function with argument(s) of any type. The **FUNCTION** statement has the general form:

```
CHARACTER*len FUNCTION func(arglist)
```

Where:

- *func* is the function name
- *arglist* is an argument list which may or may not be present
- If the form (*) is used for *len*, the length of the result returned by the function depends on the length specified for the name in the calling program unit.

The following example is of a character function which returns a number of characters of the alphabet starting at a position specified by its argument

```
CHARACTER*(*) FUNCTION SLICE(N)
INTEGER N
CHARACTER*26 ABET
DATA ABET/'ABCDEFGHIJKLMNOPQRSTUVWXYZ' /
IF (N.LT.1.OR.N.GT.26)
+ CALL ERROR ('N out of range')
SLICE = ABET(N:N+LEN(SLICE)-1)
END
```

This function could be called as follows:

```
CHARACTER*3 SLICE
PRINT *,SLICE(1),' ',SLICE(24)
```

In this case, the length assumed for `SLICE` would be 3 and the `PRINT` statement would output

```
ABCXYZ
```

`SLICE` could, of course, be used to return character values of a different length by declaring it appropriately in any calling program.

Characters as dummy and actual arguments

Character entities may appear as arguments to subroutines and functions. Whilst the basic rules for dummy and actual argument association are the same as those for arguments of other types, there are a number of rules which apply specifically to arguments of type character which come about because every character entity has a length associated with it. This length must either be specified explicitly or as `*(*)` in the subroutine or function. The table below gives the rules for dummy and actual argument association.

Dummy argument	Actual argument
Variable	Variable Array element Substring Character expression
Array	Array name Array element Array element substring

For a dummy argument which is a variable, if *dl* is the length of the dummy argument and *al* is the length of the actual argument, then *dl* must not exceed *al*. If *dl* is less than *al*, then the leftmost *al* characters of the actual argument are associated with the dummy argument. A length of `*(*)` can be specified for the dummy argument to ensure that the lengths *al* and *dl* will always be the same for that argument. Suppose that `SIMPLE` had been defined as follows:

```
SUBROUTINE SIMPLE (C1,C2)
  CHARACTER C1*8,C2*(*)
```

The following program fragment shows the effect of some calls of `SIMPLE`:

```
PROGRAM MAIN
  CHARACTER A*8,B*6,C*10
  . . .
C   argument C2 of simple assumed length 6
    CALL SIMPLE(A,B)
  . . .
C   invalid CALL : length of actual argument
C   B is less than declared length of C1
    CALL SIMPLE(B,A)
  . . .
C   the first 8 characters of C are used
    CALL SIMPLE(C,A)
  . . .
```

```
C  the actual arguments must not
C  be changed by this statement
      CALL SIMPLE('ABCDEFGH',A//B)
      . . .
      END
```

For a dummy argument which is an array, the length of the dummy argument *dl* is defined to be the length in characters of the entire dummy array. The length of the actual argument *al* is defined as follows:

Actual argument type	Length in characters
Array name	The length of the entire array
Array element	The length of the array from the element to the end of the array
Array element substring	The length of the substring

It is not essential for the declared character length of the dummy array element to be the same as that declared for the corresponding actual argument array. If a length of **(*)* is declared for a dummy argument array, the length assumed for the actual array is as follows:

Actual argument type	Length assumed for dummy declaration of <i>*(*)</i>
Array name	Length of the actual array
Array element	Length of element
Array element substring	Length of the substring

Suppose that subroutine **MESSY** had been defined as follows:

```
      SUBROUTINE MESSY (A1,A2)
      CHARACTER A1(10)*3,A2(6)*(*)
      . . .
      END
```

The following program fragment shows the effects of some calls of **MESSY**:

```
      PROGRAM MINE
      CHARACTER A(10)*3,B(20)*6,C(3,2)*5
      . . .
C  straightforward association
C  the first 6 elements of B are used
      CALL MESSY (A,B)
      . . .
C  the whole of C (30 characters) is
C  associated with A - the length assumed
C  for dummy argument A2 is 3 (the substring length)
```



```
C      and thus only the first 3 elements of B are used  
      CALL MESSY (C,B(1)(1:3))  
      . . .  
      END
```

Character entities in common blocks

A common block must either contain character data or non-character data. The Fortran 77 Standard does not allow the two types to be mixed in one common block. FTN77 allows character and non-character data to be mixed in common blocks if the /ANSI compile-time option is not used.

Language extensions

FTN77 was used for its own development. As a consequence, in order to be able to produce executable code that is optimal in core requirements and execution speed, a number of language extensions are made available by the compiler. Other extensions are provided to aid the porting of programs from other systems. It is emphasised that these extensions are not part of the ANSI Standard and their use is likely to result in non-portable programs.

The list of extensions below is only available if an ANSI directive has not appeared in the program source and a /ANSI compile-time option has not been specified.

- ☐ Long and short integer and logical data and DOUBLE COMPLEX data
- ☐ Data initialisation in type statements
- ☐ Hollerith data
- ☐ Use of @, \$ and _ characters in names and common block names
- ☐ Long names
- ☐ Octal, hexadecimal and binary values
- ☐ WHILE statements
- ☐ DO WHILE statements
- ☐ END DO statements
- ☐ Extra intrinsic functions and subroutines
- ☐ Internal procedures
- ☐ In-line 32-bit assembler
- ☐ Numeric checking of variables and array elements
- ☐ Special form of the DATA statement
- ☐ Conditional compilation
- ☐ Input/output extensions (see page 132)

- IMPLICIT NONE
- Interrupt subroutines

INTEGER and LOGICAL data types

The ANSI standard specifies that integer and logical variables should occupy the same number of storage elements as real variables but, in programs which manipulate large quantities of integer and logical data, this can be wasteful of storage space. FTN77 provides three mechanisms for controlling the storage requirements of integer and logical variables:

- 1) The default settings for a whole compilation can be chosen by using the /INTS, /INTL and /LOGS, /LOGL compile-time options.

Note that, under DOS/Win16, /INTS and /LOGS are the compiler defaults when the compiler is distributed.

To establish the defaults on your machine, issue the FTN77 command with the /HELP option.

- 2) Any combination of INTS, INTL, LOGS and LOGL must appear in an OPTIONS directive before the start of a program unit in order to specify the default for the whole of that program unit.
- 3) The following alternative forms of the INTEGER and LOGICAL statement are allowed:

```
INTEGER*1
INTEGER*2
INTEGER*4
LOGICAL*1
LOGICAL*2
LOGICAL*4
```

Note: Under Win32, INTEGER*1 parameters must not be used when declaring the size of an array.

*1, *2 or *4 overrides or confirms the current default for the program unit. For example:

```
OPTIONS(INTS,LOGS)
INTEGER*4 I,IA(10)
. . .
END
OPTIONS(INTL,LOGL)
```

```

SUBROUTINE Y(L,M)
  INTEGER*4 M
  LOGICAL*2 L,L1(3,3)
  . . .
END

```

declares the variables *I* and *M* to be four bytes in length, and the variable *L* to be 2 bytes in length. Each element of the array *IA* would occupy 4 bytes; each element of the array *L1* would occupy 2 bytes.

REAL and DOUBLE PRECISION data types

Parallel to `INTEGER*2` and `INTEGER*4` declarations there exist the alternative forms, `REAL*4` and `REAL*8`, of declarations for floating point variables. These are synonymous with `REAL` and `DOUBLE PRECISION` respectively. The size specifiers `*1`, `*2`, `*4`, and `*8` can also immediately follow individual variable names in the declaration list, overriding the length the variable would otherwise have (the syntax is analogous to that for `CHARACTER` declarations in this respect). Thus, for example, the following statements:

```

REAL*4 X*8,Y
INTEGER I,J*4

```

would declare *X* to be `REAL*8` (double precision), *Y* to be `REAL*4` (single precision), *I* to be of type default integer (set by use of `/INTS` and `/INTL` options or the `OPTIONS(INTS)` and `OPTIONS(INTL)` directives) and *J* to be of type `INTEGER*4`.

Data initialisation in type statements

It is possible simultaneously to declare and to initialise local variables and arrays as in the following examples:

```

INTEGER I/3/,J
REAL A(5)/1,2,3,4,5/
CHARACTER*2 C/'XX'/

```

The syntax for the data initialisation part of the statement is identical to that used for DATA statements except that each initialising value must immediately follow its data item and not all variables need be initialised.

Notes:

- ☐ Variables initialised in this way are assigned to static storage.
- ☐ If an array is initialised in this way, data values must be specified for the whole array.
- ☐ If a CODE/EDOC section follows an initialisation, then the CODE statement should be preceded by a CONTINUE statement.

Hollerith data and ENCODE/DECODE

Hollerith data and ENCODE/DECODE are not part of the Fortran 77 standard although they were included in Fortran 66. In FTN77 they have been implemented as extensions to the Standard. Occurrences of Hollerith data are flagged as a warning by the compiler. New programs should not use these two extensions.

Note: Hollerith editing in formats is still part of the Standard.

For example:

```
10      FORMAT (4HFRED)
```

is equivalent to:

```
10      FORMAT ('FRED')
```

Hollerith data is stored as two 8-bit characters per word, any unused character positions being blank-filled.

The number of characters contained by each type of variable is as follows:

Type	Number of Hollerith Characters
INTEGER*1 LOGICAL*1	1
INTEGER*2 LOGICAL*2	2
INTEGER*4 LOGICAL*4	4
REAL	4
DOUBLE PRECISION	8

Hollerith data is allowed in FTN77 as follows:

- 1) In DATA statements for variables and arrays of type INTEGER, REAL and DOUBLE PRECISION. For example:

```
DOUBLE PRECISION A(3)
INTEGER*4 IB(2)
DATA A/24HABCDEFGHJKLMNOPQRSTUVWXYZ/
DATA IB/8H01234567/
```

Note that apostrophes can be used as an alternative to the nH form.

- 2) As data read by READ and WRITE statements; for example:

```
      READ (1,10) I
10    FORMAT (A2)
```

An *A*w edit descriptor is used to specify that ASCII characters are to be read into or written from the specified variables which may be of type INTEGER, REAL, DOUBLE PRECISION or LOGICAL. FTN77 allows the form *A* (alone) where *w* is assumed to be the number of ASCII characters that will fit into the variable in the input/output list. For example:

```
      INTEGER*2 I
      REAL R
      LOGICAL*4 L
      READ (1,10) I,R,L
10    FORMAT (3A)
```

would read, from a single record, 2 characters into I, 4 characters into R and 4 characters into L.

- 3) In assignment statements where the left hand side is an arithmetic variable or array element, for example:

```
I = 'AB'
R = '1234'
```

Note the use of apostrophes in these examples.

- 4) In subroutine calls or function references, for example:

```
CALL PIANO(5HFORTE)
I = RS(1HA,8)
```

A run-time error will be generated when using either of the compile-time options /CHECK or /FULLCHECK if a Hollerith string is passed as an actual routine argument and the corresponding dummy argument is not of type CHARACTER.

Use of @, \$ and _ characters in names

It is often useful to be able to name a subroutine or common block in such a way that it will not clash with names chosen by everyday use of the compiler. FTN77 allows the @, \$ and _ (underline) characters as any non-initial character of a Fortran name.

Example:

```
SUBROUTINE PRINT$(I)
COMMON/COM@/A,B,C(100)
EXTERNAL F1@,F2@
INTEGER F1@
. . .
CALL MY_SUB(F1@)
. . .
```

No guarantee can be given that use of a name containing an @ character will not cause unpredictable results as a result of a clash with an FTN77 library name or other reserved name. However, no system name contains a \$ character.

Long names

FTN77 permits, as an extension, the use of variable names of up to 32 characters.

Octal, hexadecimal and binary values

Constants

An octal constant takes the form of a digit string enclosed in apostrophes and preceded by the letter O, for example:

```
O'1234'
```

The number of digits as well as the magnitude determines the length (INTEGER*2 or INTEGER*4) of the constant so that, for example, the above constant is an INTEGER*2 constant (as long as the short integer default is in use), but:

```
O'7777777'
O'0000007'
```

are both INTEGER*4 constants.

Hexadecimal constants consist of a string enclosed in apostrophes and preceded by the letter Z, for example:

```
Z'FF'
```

The length of the constant is determined by the number of digits so that, for example, with the short integer default in operation, the above is an INTEGER*2 constant, but:

```
Z'0000FF'
```

is an INTEGER*4 constant.

Binary constants consist of a string of 0's and 1's enclosed in apostrophes and preceded by the letter B, for example:

```
B'101101'
```

The number of binary digits, and whether the short or long integer default is in operation, determine whether the constant is INTEGER*2 or INTEGER*4.

Input and output

FTN77 provides O, B and Z edit descriptors for the input and output of octal, binary and hexadecimal values. The edit descriptors have the following general form:

```
Ow.m  
Bw.m  
Zw.m
```

The effect of O, B or Z edit descriptors parallels that of the Iw.m descriptor. *w* is the field width, and *m* is the minimum number of digits that must be output. *m* can be omitted if desired - the default value for *m* is 1. *m* must not be greater than *w* and has no effect for input.

The list item corresponding to an O, B or Z edit descriptor must be of type integer. For example:

```
C  read a 4-digit octal value  
      READ (1,10)I  
10  FORMAT (04)  
C  write a 10-digit binary value and  
C  include all leading zeros  
      WRITE (5,20)K  
20  FORMAT (B10.10)
```

Octal, hexadecimal and binary constants can appear instead of integer constants as items for list-directed input. The allowed forms that these constants may take are as described for constants (see page 182).

WHILE statement

FTN77 offers a WHILE statement as an alternative to an IF ... GOTO construct or a DO statement. Its general form is

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

The WHILE-block may contain any executable Fortran statements. WHILE-blocks may be nested within each other or within IF-, ELSE-, and ELSEIF-blocks and/or DO statements. The rules of nesting are the same as those for the block-IF statement.

An ENDWHILE statement can be labelled but may only be referenced from within the WHILE block.

Example:

```
J = 0
WHILE(I.NE.0) DO
  I = LIST(I)
  J = J+1
ENDWHILE
```

DO WHILE statement

FTN77 offers a DO WHILE statement as an alternative to the DO statement. Its general form is

```
DO WHILE (logical expression)
.....
END DO
```

The DO WHILE-block may contain any executable Fortran statements. DO WHILE-blocks may be nested within each other or within IF-, ELSEIF- and ELSE-blocks, WHILE-blocks and/or DO statements. The rules of nesting are the same as those for the block-IF statement.

An END DO statement can be labelled but may only be referenced from within the associated DO WHILE block.

Example:

```
J = 0
DO WHILE(I.NE.0)
```

```
      I = LIST(I)
      J = J+1
END DO
```

END DO statement

FTN77 offers an END DO statement as an alternative to the usual form of the Fortran DO statement, which requires a terminating label. Its general form is

```
DO <do-var>=<initial>,<final>,<step>
.....
END DO
```

The DO/END DO-block may contain any executable Fortran statements. DO/END DO-blocks may be nested within each other or within IF-, ELSEIF- and ELSE-blocks, WHILE-blocks and/or standard DO statements. The rules of nesting are the same as those for the block-IF statement.

An END DO statement can be labelled but may only be referenced from within the associated DO block.

Example:

```
      K = 0
DO I=1,10
      J = LIST(I)
      K = K+1
END DO
```

Extra subroutines and intrinsic functions

A number of intrinsic functions have been provided which are not in the Standard. These are defined together with the ANSI intrinsic functions in chapter 11.

Internal procedures

FTN77 provides internal procedures to allow even a few lines of coding to be used as a “subroutine” without the run-time overhead that a **CALL** statement produces.

Internal procedures have been implemented in a way which is straightforward to use and yet makes it easy to replace any internal procedure call by some standard feature such as an **ASSIGN** statement and an assigned **GOTO** statement if a program is later transported to a system which does not support internal procedures.

As its name suggests, an internal procedure is local to the program unit in which it appears. FTN77 provides four statements to deal with internal procedures:

INTERNAL PROCEDURE <list of int-proc-names>

INVOKE <int-proc-name>

PROCEDURE <int-proc-name>

EXIT <int-proc-name>

The **INTERNAL PROCEDURE** statement

INTERNAL PROCEDURE is a specification statement and must appear before any executable statement in a program unit. The general form is:

INTERNAL PROCEDURE <list of int-proc-names>

where <list of int-proc-names> is a list of internal procedure names separated by commas. Every internal procedure name used in a program unit must first appear in an **INTERNAL PROCEDURE** statement.

The **PROCEDURE** statement

The **PROCEDURE** statement is used to define the start of an internal procedure. It has the form:

PROCEDURE <int-proc-name>

An internal procedure has no argument list; any local or external name available to the program unit in which the internal procedure appears is available for use within the procedure.

Note:

When defining an internal procedure, it is up to the programmer to ensure that there is no possibility of control “flowing into” the procedure. It is suggested that internal procedure definitions are grouped together following a **RETURN** or **GOTO** statement at the end of a program unit, for example:

```

      . . .
      RETURN
      PROCEDURE P1

      . . .
      PROCEDURE P2

      . . .
      END

```

The EXIT statement

The EXIT statement is used to exit from an internal procedure. It may appear anywhere in an internal procedure definition and takes the form:

```
EXIT <int-proc-name>
```

An EXIT statement can appear wherever an executable statement is allowed (for example, at the end of an IF statement).

The only effect of an EXIT statement is to transfer control to the statement following the INVOKE statement used to invoke the internal procedure.

More than one EXIT statement can appear in an internal procedure definition. The EXIT statement need not necessarily be the last statement in a definition so that remarks made at the end of the previous section again apply here.

A program may also exit an internal procedure by executing a RETURN statement, which also leaves the parent routine.

The INVOKE statement

The INVOKE statement is used to “call” an internal procedure. It has the general form:

```
INVOKE <int-proc-name>
```

and can appear anywhere that an executable statement is allowed. The only effect of an INVOKE statement is to transfer control to the specified internal procedure.

Example of the use of an internal procedure

```

INTERNAL PROCEDURE ERROR
. . .
N = 7
INVOKE ERROR
. . .
N = 80
INVOKE ERROR
. . .

```

```
PROCEDURE ERROR
IF (N.LT.50) THEN
  CALL ERROR1(N)
ELSE
  CALL ERROR2(N)
ENDIF
EXIT ERROR
. . .
END
```

In-line 32-bit assembler

This feature of the compiler is fully described in chapter 15.

Numeric checking of variables and arrays

It is possible to specify an upper and lower limit for the value assigned to any variable or array element by means of an extension to the syntax of the INTEGER, REAL and DOUBLE PRECISION statements as in the following examples:

```
REAL A[1.0:100.0]
```

specifies that the variable *A* can take values in the range 1.0 to 100.0.

```
INTEGER I[-1:+1]
```

specifies that the variable *I* can take values in the range -1 to +1.

```
INTEGER K(10)[0:10]
```

specifies that the elements of *K* can take values in the range 0 to 10.

The general form of the limit check is:

```
[<lower>:<upper>]
```

where <lower> and <upper> are arithmetic constant expressions. Conversion takes place to the type of variable or array name, as appropriate. Thus the following are equivalent:

```
REAL A[1:10]
REAL A[1.0:10.0]
```

Arithmetic constant expressions can, of course, include **PARAMETER** names making the feature very flexible, in addition to its being simple to use.

The variable or array whose range is being checked can be local, common or an argument.

If a range check is present it is always processed by the compiler but it is only used if a **/CHECK** or **/FULLCHECK** is in force. Errors can be detected by the range check either at compile-time or at run-time.

An error will be detected at compile-time for an assignment whose right hand side is constant. For example:

```
REAL A[5.0:12.0]
A = 20.0
```

would result in a compile-time error.

Care should be taken when using real and double precision range checks to allow for the effects of rounding error. For example:

```
REAL A[2.2:5.275]
A = 3.1
A = A+2.175
```

might generate a range check error. The upper limit should be specified as 5.2751.

An error will be detected at run-time for all other assignments and statements such as **DO** and **READ** which imply assignments. Such errors produce the message:

```
*** User-specified range check error
```

Note:

Statements such as:

```
DO 10 I=1,20
```

imply a final assignment to **I** of the value 21. Any range check used for **I** should take this fact into account.

Special form of the **DATA** statement

This facility, which permits **INTEGER*4** variables to be given address values, is described on page 199.

Conditional compilation

FTN77 provides conditional compilation by means of the **SPECIAL PARAMETER** statement, the **/SPARAM** compile-time option and the **CIF**, **CELSE** and **CENDIF** statements.

SPECIAL PARAMETER and /SPARAM

The specification statement

SPECIAL PARAMETER <name>

defines <name> to be of type integer. <name> must not appear in a type statement and is local to the program unit in which the **SPECIAL PARAMETER** statement appears. The value represented by <name> is set by means of the **/SPARAM** compile-time option as follows:

/SPARAM <integer>

where <integer> is the required value.

Any number of **SPECIAL PARAMETER** names are allowed per program unit but they are all assigned the same value.

CIF, CELSE and CENDIF

CIF, **CELSE** and **CENDIF** are used to select the statements in a program unit that are to be used during a particular compilation. Their general forms are:

CIF (<name>.EQ.<constant>) **THEN**

...

CELSE

...

CENDIF

where <name> is a **SPECIAL PARAMETER** and <constant> is an integer constant. **CIF** etc. begin in column 7 or after.

CIF and **CENDIF** must appear in pairs: their appearance constitutes a **CIF-block**.

The actual value assigned to the special parameter <name> is compared with the integer constant <constant>. If the two agree, the statements following **CIF** are compiled until a **CELSE** or **CENDIF** statement is found.

If the two disagree, statements are ignored until a **CELSE** or **CENDIF** statement is encountered. Such statements are denoted by a back-slash character in the listing file.

Once **CENDIF** is encountered, the **CIF-block** is complete. **CELSE** causes the reverse effect to that specified by the preceding **CIF** statement.

CIF..CENDIF blocks can be nested and CELSEIF may be used to replace the sequence ELSE, CIF,.....CENDIF as in the Fortran 77 IF-statement.

IMPLICIT NONE

This is a compiler directive that causes the compiler to fault the subsequent use of a variable which has not been given an explicit type. It appears in column 7 or afterwards and should be placed in every subprogram where it is needed and before all executable statements in that subprogram.

An alternative is to use `OPTIONS(IMPLICIT_NONE)` at the head of a file and this will then apply to the whole file. Another alternative is to configure the compiler using the compiler option `/CONFIG` and to set `IMPLICIT_NONE` as the compiler default.

INTERRUPT subroutines

The `SET_TRAP@` routine enables a program to catch a number of program events (see the *FTN77 Library Reference* manual or the on-line Help system for details of this routine and other routines mentioned in this section). Certain events, such as the QUIT trap, can interrupt a program at an arbitrary point. The routine which is used to catch such an interrupt must be specially written to cater for this. On page 206 an assembler technique is described for this purpose. While this technique offers the greatest flexibility, the `INTERRUPT SUBROUTINE` offers a simpler mechanism which is normally adequate.

An interrupt subroutine must have no arguments and is normally terminated by stopping (`STOP` or `CALL EXIT`) or by calling `JUMP@` to return to a label in an earlier routine. The following program illustrates this technique:

```

      EXTERNAL QUITH
      INTEGER*4 X
      COMPLEX*16 LABEL
      COMMON LABEL
      CALL LABEL@(LABEL,*1)
      CALL SET_TRAP@(QUITH,X,0)
1     READ *,N
      DO 2 I=1,N
2     PRINT *,N

```

```
PRINT *, 'END OF LOOP'
GOTO 1
END
INTERRUPT SUBROUTINE QUITH
COMPLEX*16 LABEL
COMMON LABEL
CALL COU('Quit trapped!')
CALL JUMP@(LABEL)
END
```

By pressing **Ctrl Break** you can force this program to abandon the loop in progress, print a message, and return to read more data. Clearly this technique is of great value in writing interactive programs.

It is possible on return from an **INTERRUPT SUBROUTINE** to continue from the point at which execution was interrupted, provided that the subroutine neither performs Fortran input/output calls nor any other system routine. For example, such a subroutine might simply set a flag in a common block and return to the calling program.

If the **INTERRUPT SUBROUTINE** either calls a system routine or performs input/output, and the interrupt takes place from within a system routine or the Fortran input/output system, then unpredictable effects can result after return from the routine. However, control can be passed back to the program using the **LABEL@** and **JUMP@** routines.

The in-line assembler

Introduction

This chapter explains how to write 32-bit assembler instructions in Fortran programs. It may be omitted by readers who have no interest in the details of the Intel microprocessor environment. FTN77 users wishing to code at the assembler level should obtain a copy of a Programmer's Reference Manual published by Intel. For details of the DBOS execution environment see page 314.

The execution environment (Win32)

Each process executes in its own 32-bit virtual address space. This gives 2Gbytes for the combined code and data spaces (the remaining 2Gbytes are reserved for the operating system). Using the advanced features of the 486 chip, each process address space is protected from modification by other processes executing within the system.

The CODE/EDOC facility

The CODE statement switches the compiler into a mode in which it accepts Intel 32-bit assembler instructions rather than Fortran statements. The compiler is returned to normal by the EDOC statement. A CODE/EDOC sequence may appear anywhere that an executable Fortran statement is permitted. For example:

```
CHARACTER*10 L
CODE
```

```
LEA    EDI%,L        ;EDI gets address of L
MOVb   AL%,='*'      ;Asterix in AL
MOV    ECX%,=10       ;Count in ECX
REP                                ;Rep prefix coded as
+                                ; a separate instruction
STOSB                                ;This fills L with asterisks
JMP     $10            ;Jump to label 10
EDOC
PRINT *, 'This should not be printed'
STOP
10     PRINT *, 'L = ',L
END
```

This example is artificial in that there is no real point in performing operations in assembler that can be done in Fortran, however it illustrates that code is written according to the following conventions:

- ☐ Instructions refer to Fortran objects or explicitly to the registers
- ☐ Register names are followed by a '%' to distinguish them un-ambiguously from variable names.
- ☐ Instructions must start in column 7 or beyond.
- ☐ Only numeric (Fortran) labels are permitted.
- ☐ Comments may be included provided they are preceded by a semi-colon character (;)
- ☐ Some mnemonics are followed by an 'H' to indicate halfword (16-bit) operation or by 'B' to indicate a byte operation. This is discussed in more detail below.

Mixing of Intel 32-bit Assembler and Fortran

Assembler programs should not alter registers EBX% (pointer to local static data), EBP% (pointer to local dynamic data), or ESP% (stack pointer). Other general registers can be used freely.

Under DOS/Win16, in certain cases (notably in conjunction with SVC's) it may be necessary to alter EBP% or EBX%. In this case the contents should be pushed prior to the operation and restored afterwards with a pop instruction.

The coprocessor will be empty and in rounding mode when control is passed to assembler, and it must be in the same state afterwards. It is possible to jump from Assembler to labelled Fortran statements and vice-versa.

Labels

As in the above example, labels are Fortran labels and are referred to by preceding the numeric label by a dollar character thus:

```
JMP $7
```

Conditional jumps are coded in 32-bit form when necessary, so these may be used without considerations of range. The **LOOP** instructions, which do not have 32-bit forms, are not supported by the assembler.

Referencing Fortran variables

Variables are referenced using the following scheme.

- Local dynamic variables are addressed using **EBP%**.
- Local static variables are addressed using **EBX%**.
- An argument can only be referenced by its address. For example, in order to load the argument **L** into **AX%** use:

```
SUBROUTINE FRED(L)
INTEGER*2 L
CODE
  MOV      EAX%,L           ;Get address of L
  MOVH     AX%,[EAX%]       ;Load halfword
  .
  .
```

- References to common or external variables are constructed using a full 32-bit address.

These rules mean that local variable references may be indexed by one extra register, and common variables may have two indexing registers if necessary. For example:

```
INTEGER*4 L(200)
CODE
  MOV      EAX%,8
  MOV      L[EAX%],0        ;This sets L(3) to 0
```

Variable references may be offset. For example:

```
CHARACTER*10 F00
CODE
  MOVB      F00+3,=32      ;Sets F00(4:4)=' '
```

Indices can also contain multipliers of 1 (default), 2, 4, or 8. For example:

```
ADD      [EAX%+ECX%*4], =6
```

Literals

An instruction operand may be a constant (literal). In this case the constant must be preceded by an '='. Floating point instructions may have literal arguments, which are placed in memory and addressed, since there is no immediate form of these instructions.

Literals may contain any constant expression (which will be evaluated using the standard Fortran 77 rules) and may be of any type. For example:

```
FLD      =5.0
DFADD    =5.0D0      ;Note REAL*8
                        ; constant needed
MOV      EAX%,=(4*5)  ;Load 20 into EAX%
TEST     FRED,=Z'FEFFFFFF' ;Hex constant
```

Halfword and byte forms of instructions

In standard (16 bit) assembler notation, many instructions have two forms depending on whether the operand is of type byte or word. In 32-bit assembler, instructions may have three forms - full word (32-bit), half word (16-bit) and byte (8-bit).

Rather than follow the Intel convention that the instruction is defined by its operand (something which is hard to define in the context of Fortran variables), each distinct instruction has a different mnemonic. The conventional Intel mnemonic refers to the 32-bit form of the instruction, and we append an 'H' to refer to a half word instruction (constructed using an operand size prefix) or a 'B' to refer to a byte instruction where available. Thus for example we have the following string move instructions:

```
MOVVS      ;Move a full word
MOVSH      ;Move a half word
MOVSB      ;Move a byte
```

A similar scheme is used with the memory reference coprocessor instructions. Thus we have for example four types of memory reference floating point additions:

```
FIADDH    I            ;Add an integer half word
FIADD     L            ;Add an integer full word
FADD      R            ;Add a short real (4 bytes)
DFADD     D            ;Add a long real (8 bytes)
```

Using the coprocessor

Coprocessor stack operands are referred to using the following notation:

```
ST(0)          ;Stack top
ST(1)          ;Next to stack top
etc.
```

Stack reference instructions use the short real form of the mnemonic (for example, FADD) but the actual calculations are performed to the full precision of the coprocessor. The coprocessor stack must be returned empty and with the control word unchanged. Coprocessor instructions do not contain an implied WAIT. WAIT instructions are only necessary after results are returned to the 32-bit Intel chip (FSTP, or FSTSWAX for example), and even then only if the result will be used before another coprocessor instruction is started.

If you are writing code that might be used in an another environment, you should ensure that any coprocessor mnemonics you use are appropriate to that environment.

Under DOS, Weitek mnemonics are supported for operations for the 1167 and 3167 coprocessors on a 386 and for operations for the 4167 coprocessor on a 486.

Instruction prefixes

The following prefixes are available as pseudo instructions coded on the line above :

```
REP
REPE
REPNE
FS          ;Appends the FS: prefix
GS          ;Appends the GS: prefix
```

For example:

```
FS           ;Source operand from FS
REP
MOVSB
```

or

```
MOV FS:0,ESP%
```

would be coded as

```
FS
MOV 0,ESP%
```

Other prefixes are rarely needed, however they are available using the **DB** pseudo instruction to code an arbitrary byte. For example the **CS** prefix could be coded as:

```
DB          Z'0E'
```

Other assembler facilities

In general, the assembler pseudo-instructions and macros are not available, as equivalent or more powerful facilities are available using FTN77. The following pseudo instructions have been provided:

- In-line data may be inserted using **DB**, **DW**, or **DD** pseudo instructions. For example:

```
DD    Z'FFFFFFFF'      ;4 bytes
DW    45                ;2 bytes
DB    -1                ;1 byte
```

- Under DOS/Win16, the **SVC** pseudo instruction has been provided to facilitate calls to **DBOS**. The only **SVC** calls of general interest are **SVC/3** and **SVC/26**. **SVC/3** is described in detail below; **SVC/26** is used to set the **IOPL** (I/O permission level) of the program. If **EAX%=1** user I/O is enabled, if **EAX%=0** user I/O is inhibited (default). After using this **SVC** it is possible to use **IN** and **OUT** instructions to control peripherals (or crash the machine if you are not careful!).

- Do not try to select the coprocessor rounding mode by using an explicit **FLDCW**, as this will invalidate the independent control of arithmetic precision. Each of the following pseudo instructions is snapped on first use to the appropriate **FLDCW** command referencing a table of suitable control words:

```
FROUND      ; Select rounding mode
FCHOP       ; Select chopping towards 0
```



```

FCHOPM          ; Select chopping towards - infinity
FCHOPP          ; Select chopping towards + infinity

```

Calling MS-DOS and BIOS

See page 316.

Other machine-level facilities

It is always inconvenient to have to descend to assembler, even in the form of a CODE/EDOC sequence, and a number of special Fortran constructions have been introduced for convenience.

- The intrinsic functions CCORE1, CORE1, CORE2, CORE4, FCORE4 and DCORE8 are available to examine the contents of a given location.

Each function takes an INTEGER*4 argument. CORE1 returns the INTEGER*1 byte at that address, CORE2 returns the INTEGER*2 word at that address and CORE4 the corresponding INTEGER*4 value, FCORE4 the corresponding REAL value and DCORE8 the corresponding DOUBLE PRECISION value. CCORE1 returns a single byte (as a character).

These functions must be declared in an INTRINSIC statement before they are used. They may also be used on the left hand side of an assignment. For example:

```

CORE2(L) = CORE2(L)+1
CCORE1(P)=' '
CORE1(PTR)=123

```

If an argument to a routine is one of these functions *the actual address is passed*, for example:

```

INTEGER*4 L
INTRINSIC LOC,CORE2
K = 4
L = LOC(K)
CALL FRED(CORE2(L))
PRINT *,K
END
SUBROUTINE FRED(M)
M = M + 2
END

```

would print 6.

- A special form of the **SUBROUTINE** statement is available thus:

```
SPECIAL SUBROUTINE JACK
```

Special routines must have no arguments, and contain no preamble to set **EBX%**, **EBP%** etc. They can only really be followed by **CODE/EDOC** sequences, and no reference to dynamic variables must be made in such a routine. Static variables may be referenced and will use the full address form of the instruction (rather than **EBX%** relative). Special subroutines may contain additional entry points coded as special entries:

```
SPECIAL ENTRY BILL
```

Special routines may not contain ordinary entry points and vice-versa. The return from a special subroutine must be via a **RET** instruction and not as a result of executing a **RETURN** or **END** statement. The main purpose of the special subroutine is as a routine which can be called from assembler without altering the contents of the registers.

An additional use of this facility is in conjunction with the **SET_TRAP@** routine. A control break or floating point fault can take place at an arbitrary point in a program, and it is important to be able to save the registers etc. before they are overwritten. Although this can be done with an interrupt subroutine without the use of **CODE/EDOC**, the latter offers the ability to inspect and alter the contents of the registers if desired.

- **INTEGER*4** variables may be given *address values* in **DATA** statements. For example:

```
DATA L/%K/
```

would give **L** the value of the address of **K**. The address must be of a local static, external or common variable.

- Circular shifts are available as intrinsic functions and thus do not require the use of assembler.
- The **LOC** intrinsic function returns the address of its argument as a 32-bit number.

Error messages

Owing to the syntax of assembler, use of unpaired apostrophes and parentheses in comments in **CODE/EDOC** sequences will cause the compiler to output apparently spurious messages concerning the mismatching.

The in-line assembler and DBOS

FTN77 programs and the DBOS environment

This chapter gives some information about what basic forms of instructions are used to access different storage classes of data, and also discusses the subroutine linkage conventions used by FTN77. This will help both with recognition and understanding of some of the instruction sequences emitted by FTN77, and also with writing CODE..EDOC in-line assembler sequences under DOS/Win16.

Segment selector registers

The segment selector registers CS, DS and SS (code, data and stack segments respectively) are set up to point to the entire virtual address space with no offset (i.e. they all point to virtual address zero with a 4-gigabyte segment limit).

The ES selector is used by some string instructions which require it. The FS selector is not dedicated for any particular use, and so it is available for temporary use for special applications (for example, see the description of the use of the DOSCOM@ routine in the *FTN77 Library Reference* manual). The GS selector is set up to point to the memory space mapped onto the Weitek coprocessor when one is installed.

Variable storage

FTN77 programs make use of the EBP% register to address local dynamic variables.

Scalars are allocated at positive offsets from EBP%, while arrays are allocated at negative offsets. This arrangement is used so as to maximise the number of objects (and hence hopefully the number of instructions) which require only a one-byte offset to access them. For example, if an array of size more than 128 bytes were allocated at offset 0 from EBP%, then no other objects allocated relative to EBP% could be referenced with a one-byte offset, thus effectively wasting some code space on instructions which would then need multi-byte offsets.

When not operating in /OPTIMISE mode, the EBX% register is used to point to the static data space (i.e. saved variables). Scalars are allocated at negative offsets from EBX%, again to increase the number of objects which can be referred to with short offsets. Note that the scheme can be used either way around. The range of offsets which are accessible with one byte is -128 to 127, since the offset is treated as signed. FTN77 does not use a more optimal allocation strategy due to the constraints of its single-pass nature.) Where constants cannot be compiled into immediate mode instructions (for example, where they appear as subroutine arguments), they are placed relative to EBX%.

When /OPTIMISE is in effect, the EBX% register is not used in this way. All references to static objects are planted as absolute address references (which are of course re-located appropriately by the linker). In this way the EBX register becomes free to be used for more general purposes. When optimisation is in effect the availability of another register is especially beneficial, due to the extra sophistication of the register allocation algorithms used.

Thus, the instructions output for an assignment of a static short integer scalar to a dynamic short integer scalar might be of the following form:

```
MOVH      CX%,DS:[EBX%-02]
MOVH      SS:[EBP%+08],CX%
```

(Note that the syntax used here for segment register references is not directly acceptable to the FTN77 in-line assembler. In fact, since DS% and SS% point at the entire real mode address space, it would not be necessary to specify them explicitly when assembling instructions such as those above.)

Arguments to routines are passed as pointers, which are stored relative to EBP% in the initialisation code for the routine (usually from offset 0 onwards for a main entrypoint). Thus the code to load a short integer argument into the AX% register might be of the following form:

```
MOV      ECX%,SS:[EBP%+04]
MOVH     AX%,DS:[ECX%]
```

Note that even when referring to an object which is an argument by name in an in-line assembler sequence, it is necessary to perform the de-referencing described above. Failure to do so will result in the error message "Illegal memory reference" at compile time.

Common variables (and static variables when /OPTIMISE is in effect) are fixed up to an absolute address by the linker, and thus are not referenced relative to any base register. A common reference might look something like the following:

```
MOVH     AX%,DS:[+00020100]
```

Linkage to subroutines

There are three aspects to this topic - the code to call a subroutine, the code executed at on entry to the subroutine, and the code to return from the subroutine. The scheme has to achieve four things:

- Transfer of the arguments to the procedure.
- Setting up the required value of **EBX%** for the called routine's static data space, and reinstating its value on return.
- Setting **EBP%** to point to a new "stack frame", or local dynamic data space.
- Transfer of control to the procedure and return to the point immediately after the procedure invocation afterwards.

Of course, the most natural mechanism to use for the last of these requirements is the **CALL** instruction, together with a corresponding **RET** instruction. This implies that the system stack, with stack pointer **ESP%** is used, so the value of **EBP%** is set up with respect to **ESP%**.

As an example, we will look at a call to a subroutine which takes two short integer arguments. The code to call the subroutine is as follows:

```
PUSH      0000009E
LEA       EAX%,SS:[EBP%+10]
PUSH      EAX%
MOV       ESI%,ESP%
CALL      000000A0
ADD       ESP%,00000008
```

The first **PUSH** puts a pointer to the second of the two arguments onto the stack (arguments are always pushed in reverse order). In this example, since the argument is an absolute address, we can deduce that the argument in question is either a static or common variable, or an external. The second **PUSH** is of a local dynamic variable - its address has to be obtained at runtime by the **LEA** instruction since, unlike static and common variables, its address cannot be determined at link time.

Next, the **ESI%** register is loaded with the value of the stack pointer **ESP%**, and the routine is called. Thus on entry to the called routine **ESI%** contains the address of the start of the argument pointers, which appear in order in ascending memory address (since the stack builds downwards as values are pushed onto it). Upon return, the **ADD** instruction effectively pops the argument pointers off the stack (the value 8 reflects the two 4-byte argument pointers).

For the case of character arguments, a 32-bit length is passed for each of the character type arguments in the argument list, and these appear after all of the pointers for the actual arguments in the argument list (that is, they are **PUSHed** onto the stack before the actual arguments to the procedure).

On entry to the subroutine, code such as the following is planted:

```

PUSH      EBX%
PUSH      EBP%
SUB       ESP%,00000044
LEA       EBP%,SS:[ESP%+30]
LEA       EBX%,DS:[+E0]
MOV       ECX%,=00000002
LEA       EDI%,SS:[EBP%]
REP       MOVs

```

First of all, **EBX%** and **EBP%** are pushed onto the stack, so that they can be returned to the required values for the caller on return. Next the stack pointer, **ESP%**, is moved down sufficiently to make space for the called procedure's stack frame, and **EBP%** is set up relative to this new value of **ESP%**. In the example, the called procedure has a 20-element local dynamic short integer array, and one local short integer scalar, and two arguments. The allocation relative to **ESP%** is as follows (offsets in hex):

```

EBP%-28 to EBP%-01    array
EBP%+00 to EBP%+03    argument pointer for first argument
EBP%+04 to EBP%+07    argument pointer for second argument
EBP%+08 to EBP%+09    local short integer scalar

```

To write **CODE..EDOC** in-line assembler sections it is not strictly necessary to understand the storage allocation scheme beyond a very general level. To access any given object in an instruction, it can be specified by name, and its address is fixed up by the compiler, possibly with the help of the linker, in the same way as it would be dealt with in a Fortran statement. For those that are interested, the actual storage allocation of variables is given in the listing produced by the **/MAP** compilation option.

Next the **EBX%** register is fixed up to point to the local static data space (this instruction is fixed up by the linker when the space is allocated - the object file specifies how big a space is needed, and the linker allocates the requested amount of space).

The remaining instructions deal with copying the argument pointers from the location given by **ESI%** (set up prior to the **CALL** instruction), to a known offset in the stack frame, usually from offset zero relative to **EBP%**. The case where this may not occur is when a procedure has entrypoints, and the same arguments may appear in different positions in the argument list. However, as far as possible contiguous strips of argument pointers are maintained (so that these can be copied by instructions with the **REP** prefix). It is at this point that any character lengths are dereferenced, and their values stored in a space allocated from the called procedure's local dynamic storage.

If extra arguments are present on the call, their corresponding pointers are not copied (the number of fullwords copied is equal to the number of arguments specified in the subroutine declaration). If too few arguments are supplied then the pointers for those not present are given by undefined memory locations, and attempts to access these may simply give meaningless arguments, or may cause a general protection exception, or for floating point quantities may cause a coprocessor fault.

However, if both the calling and the called procedure are compiled with the `/CHECK` or the `/FULLCHECK` compiler options, then the mismatch in the number of arguments is picked up by the argument checking mechanism. (We will not describe the checking mechanism in any detail here, since it is assumed that if the program being debugged has been compiled with one of the checking options then it does not violate any of the rules.)

The code to return to the calling procedure simply restores the value of the stack pointer `ESP%` to its value at the start of the procedure just after the callers local static and dynamic space pointers have been restored, pops these from the stack, and then executes a `RET` instruction:

```
LEA      ESP%,SS:[EBP%+14]
POP      EBP%
POP      EBX%
RET
```

Functions are called in the same way as subroutines, except that the function value for each function type is returned according to the following conventions:

INTEGER*2	AX%
INTEGER*4	EAX%
LOGICAL*2	AX%
LOGICAL*4	EAX%
REAL*4 and REAL*8	ST(0)
COMPLEX*8 and	Real part in ST(0) and
COMPLEX*16	imaginary part in ST(1)

Functions which return a character type result are called by a slightly different mechanism. Before calling the function, the register `EDI%` is set up to point at the destination for the function value (note that this may be a compiler-generated temporary variable). All assignments to the function value in the called routine go through a copy of this pointer in the called procedure's local dynamic space, and so directly affect the intended operand. The declared character length of the function is `PUSHed` before other lengths associated with character-type arguments, for use by `CHARACTER(*)` functions.

There are a number of other aspects of the generated code which are not described here (for example, the argument checking mechanisms mentioned earlier in this

chapter). However much of any information which might be required can be deduced from the expanded listing generated by the /EXPLIST compiler option.

Trap routines

DBOS can simulate an interrupt if certain conditions occur. This can be enabled with the SET_TRAP@ routine. See the description of SET_TRAP@ in the on-line Help system or the *FTN77 Library Reference* manual for a list of the conditions which can currently be trapped.

To use this feature the trap routine must save and restore the registers bearing in mind that the event will usually occur between statements. Here is an example of a simple CONTROL BREAK handler:

```
EXTERNAL QUIT_TRAP
INTEGER*4 P
CALL SET_TRAP@(QUIT_TRAP,P,0)
:
:
:
SPECIAL SUBROUTINE QUIT_TRAP
CODE
    PUSHF                ;SAVE ALL REGISTERS AND FLAGS
    PUSHA
    SUB    ESP%,106;MAKE ROOM FOR COPROCESSOR STATE
    FSAVE [ESP%]
    FINIT
    FROUND
    CALL   QUIT1
    FRSTOR [ESP%]
    ADD    ESP%,106
    POPA
    POPF
    RET
EDOC
END
```

Note that this program could have been coded using an INTERRUPT SUBROUTINE (see page 191) without the need for CODE/EDOC and is used purely as a simple example. The QUIT1 routine can take any action desired except that if it uses Fortran I/O statements it must not return back into an I/O statement. (A key press could occur at any point.) The use of low level I/O routines such as COU@ is not restricted. A useful technique is to set up a label with LABEL@ and pass it in

COMMON. The QUIT1 routine can then use JUMP@ to pass control to the label. See the *FTN77 Library Reference* manual or the on-line Help system for detailed information on COU@, LABEL@ and JUMP@.

The machine code programmer's window

The window based debugging system, described in chapter 7, also offers a window for debugging at the machine code level (see page 68 for further details).

16.

Mixed language programming

Introduction

This chapter discusses the details of inter-language programming between Fortran and Salford C/C++. The sizes of the various data types, data storage and function call styles are covered in order to facilitate the mixing of modules compiled in either language.

Data types

Basic data types

The table 16-1 illustrates the amount of storage required for the basic data types associated with each language: In all the languages, pointers are represented as 32-bit quantities.

Arrays

There are two methods of storing arrays, row-wise and column-wise. Row-wise storage means that the elements are stored a row at a time starting from a base address and increasing towards high memory. Arrays stored column-wise have the elements stored a column at a time increasing towards high memory.

For example, consider the array consisting of 10 rows and 20 columns. The appropriate declarations in each language would be:

```
FTN77      INTEGER*4 numbers(10,20)
FTN90      INTEGER*(KIND=3) numbers(10,20)
C/C++      int numbers[20][10];
```

Data type	Size (bytes)	C/C++	FTN77	FTN90
Integer	1	char	INTEGER*1	INTEGER (KIND=1)
	2	short int	INTEGER*2	INTEGER (KIND=2)
	4	int; long int	INTEGER*4	INTEGER (KIND=3)
Unsigned integer	1	unsigned char	-	-
	2	unsigned short int	-	-
	4	unsigned int	-	-
Logical	1	char	LOGICAL*1	LOGICAL (KIND=1)
	2	short int	LOGICAL*2	LOGICAL (KIND=2)
	4	int	LOGICAL*4	LOGICAL (KIND=3)
Real	4	float	REAL; REAL*4	REAL (KIND=1)
	8	double	REAL*8; DOUBLE PRECISION	REAL (KIND=2)
	10	long double	-	-
Character	1	char	CHARACTER*1	CHARACTER*1

Table 16-1

A row-wise array would be stored as:

```
numbers(0,0); numbers(1,0); numbers(2,0); ... numbers(9,0);
numbers(0,1); ....
```

whilst a column-wise array would store the elements as

```
numbers(0,0); numbers(0,1); numbers(0,3); ... numbers(0,9);
numbers(1,0); ....
```

The various language standards define Fortran as using column-wise storage, whilst C/C++ stores arrays row-wise. Therefore, a Fortran array defined as `numbers(10,20)`, would have the equivalent C/C++ declaration `numbers[20][10]`.

Character strings

C/C++ character strings are stored as a NULL (character zero) terminated arrays of characters whilst Fortran character strings are fixed length and are padded to the end of the array with spaces. It is important to take into consideration these different methods of storing strings when passing or receiving them as parameters.

Calling FTN77 from C/C++

Introduction

The following text assumes that you are writing in C/C++ and are calling an FTN77 relocatable binary library (RLB) or dynamic link library (DLL).

When calling FTN77 routines from C/C++, the following major points should be considered:

- ☐ Fortran arguments are passed by reference rather than value.
- ☐ All Fortran external names are upper case (regardless of the case of the original source text).
- ☐ Fortran character variables have no simple analogue in C/C++.

If you have a C/C++ main program calling a Fortran RLB, then the main program should call the library initialisation routine if there is one. Failure to do so will result in unpredictable behaviour. If you are calling a DLL then the initialisation will probably take place automatically when the DLL is loaded.

CHARACTER variables

Fortran character arguments are fixed length and padded with space characters. In order to determine the length of a character argument, the FTN77 compiler passes the length of the string as an extra argument at the end of the argument list for the subroutine/function. If more than one character argument is passed, then the lengths are passed in the order in which the character arguments appear in the argument list. For example:

```
SUBROUTINE COMPARE(STRING1, STING2)
  CHARACTER*(*) STRING1, STRING2
  .
  .
END
```

This subroutine would have the following C/C++ prototype:

```
extern "C" COMPARE(char *s1, char *s2, int l1, int l2);
```

where *l1* and *l2* are the lengths of the two strings *s1* and *s2* respectively. In order to call **COMPARE** from within a C/C++ program, the programmer must pass the lengths of the two strings so the call would look something like this:

```
char *str1, *str2;  
.  
.  
COMPARE(str1, str2, strlen(str1), strlen(str2));
```

Arrays

As we have already noted, the standards for C/C++ and Fortran define array storage differently. It is therefore necessary to provide an interface routine between the FTN77 library and the C/C++ code or to modify the C/C++ code to take into account the differences in data storage.

INTEGER, LOGICAL and REAL

It is necessary to ensure that the data type of the C/C++ variable matches that of the FTN77 variable (see table 16-1). All parameters in the Fortran argument list will be passed by reference. You should therefore declare each argument as a pointer in C or as a reference variable in C++.

Common blocks

The FTN77 compiler automatically adds an underscore “_” character onto the end of a common block name. This is transparent to the programmer unless you wish to access the data stored within the common block. It is necessary for the C/C++ programmer to explicitly add this underscore character to the common block name before use. Alternatively, the program **COMGEN** (see chapter 17) may be used.

Calling C/C++ from FTN77 or FTN90

The **C_EXTERNAL** keyword has been added to give the Fortran programmer the added flexibility of being able to call C/C++ routines and forcing the compiler to generate extra code to handle some of the data conversions. An example of this is the string data type. As we have already noted, C/C++ strings are NULL terminated, whilst Fortran strings are fixed length, padded with spaces. The **C_EXTERNAL** declaration for a function, informs the compiler that the function or subroutine is written in C/C++ and is external to this program module. If the function uses a string

data type, code is planted to generate a C/C++ style string before entering the C/C++ function and then after the function call, code is generated to convert the C style string back into a Fortran string. This frees the C/C++ programmer from the additional complexities of providing the conversion code. It also means that a Fortran programmer can call a third party library without converting all string references into C/C++ strings before calling an external routine.

The `C_EXTERNAL` declaration has the following form:

```
C_EXTERNAL name ['alias'] [(desc , ...)] [:restype]
```

where:

name

is the name to be used to call the function in the Fortran program.

alias

is the external name used for the routine (i.e. the name that is used in the C/C++ source code).

desc

describes the arguments that the routine receives and/or returns.

restype

identifies the routine as a function and describes the type of the object returned; this may be any function type other than `CHARACTER`.

Some examples of valid `C_EXTERNAL` declarations are given below.

Example 1

```
C_EXTERNAL SUB
```

This describes an external C/C++ routine which accepts no arguments and returns no result. The corresponding C/C++ declaration would be

```
extern "C" void SUB(void)
{
    ....
}
```

and the function would be called by the Fortran statement `CALL SUB`.

Example 2

```
C_EXTERNAL WRITE 'WriteFile' : INTEGER*4
```

This describes a C/C++ routine called **WriteFile** which accepts no arguments, but returns an integer result. The routine is called from a Fortran program by the statements

```
INTEGER*4 RESULT
RESULT = WRITE()
.....
```

The C/C++ code could have the following form

```
int WriteFile(void)
{
.....
}
```

Before continuing, we must first examine the possible forms of the *desc* parameter and the *restype* part of the declaration in more detail.

The *desc* parameter allows the programmer to over ride the default linkage of arguments. If you use these argument descriptors, the number of arguments in each occurrence of a call must agree with the number of descriptors in the routine definition. *desc* may be any one of: REF, VAR, STRING, INSTRING, OUTSTRING.

The VAL specifier may only be used for numeric and logical scalars. Instead of pushing the address of the value onto the stack, the actual value is pushed. This allows C/C++ functions to use its arguments as local variables. The REF specifier may be used with any Fortran object. This forces the Fortran program to push the address of the object onto the stack. This is the default action but should be used as a matter of good programming practice to allow the compiler to check for the correct usage of external functions. So we may additionally have the following descriptions:

```
C_EXTERNAL UNIX_WRITE 'write' (VAL, REF, VAL): INTEGER*4
```

for the UNIX low-level **write** function. This is defined in C/C++ as

```
int write(int handle, void *buffer, int amount)
{
.....
}
```

but it now looks to the Fortran program as if it was a Fortran function declared as:

```
FUNCTION UNIX_WRITE(HANDLE, BUFFER, BUFSIZ)
INTEGER*4 HANDLE, BUFFER, BUFSIZ, UNIX_WRITE
```

The remaining three types, STRING, INSTRING, OUTSTRING, are a little more complicated. All three are used to describe a string object. Each one forces the compiler to do differing amounts of work before the function call is made. As we have already seen from the discussion at the start of this section, the compiler can be forced to convert strings from Fortran strings to C/C++ strings and visa-versa. This is the default action and is equivalent to the STRING descriptor. However this causes an unnecessary overhead if an argument is to be used for either input to a function or output from a function but not both. In this case the INSTRING and OUTSTRING

maybe used. This saves the redundant copy operation from taking place. It is also possible to restrict the length of the temporary variable used to store the string which is actually used in the function call. The default length of the string is the length of the CHARACTER array or 256 bytes in the case of a CHARACTER*(*) array. This is done by specifying the length of the string in parentheses after the descriptor. Further examples of the C_EXTERNAL are:

```
C_EXTERNAL COPY_STRING 'strcpy' (OUTSTRING, INSTRING): INTEGER*4
C_EXTERNAL STRNCPY 'strcat' (STRING, INSTRING(40)): INTEGER*4
```

where **strcpy** and **strcat** are the standard C library functions.

Under Win32, the syntax of the declaration for a C_EXTERNAL function is similar to a STDCALL statement (see chapter 18 for details).

Calling Windows 3.1 functions

FTN77 and FTN90 contains two further keywords, WINREF and WINSTRING. These are available to aid the writing of programs that use the Windows 3.1 API calls. The source module containing these keywords must be compiled using the /WINDOWS option.

WINREF passes the argument by reference but converts the pointer into a windows style pointer (i.e. 16-bit segment and offset) rather than a true 32-bit pointer.

WINSTRING arguments are input strings to windows functions. Again the pointer is converted from 32-bit form to 16-bit form. Further details are given in the *ClearWin+ User's Guide*.

Mixing I/O systems in C/C++, FTN77 and FTN90

In general the I/O systems in these three languages are different and should not be mixed. For example, it is not usually possible to open a file in FTN77, and then pass the handle to be used in C/C++. The only exception to this rule is that if you use DBOS library calls to manipulate files, then the handles are common across the language boundary.

The COMGEN utility

Introduction

The COMGEN utility can be used to translate a source file containing definitions of common blocks, parameters, externals and intrinsic declarations into Fortran and C/C++ include files. By using a central source file and the INCLUDE directive (see page 36), it is possible to ensure that all modules are using consistent definitions of the common blocks they require. This also provides a method of accessing Fortran common blocks as C/C++ structures.

Command line

COMGEN is invoked in the following manner:

`COMGEN source dest1 [dest2]`

where *source* contains the source declarations for COMGEN, *dest1* is the name of the file to be overwritten with the Fortran declarations and *dest2* is the name of the file which will contain the C/C++ declarations.

Source file format

The source file is broken down into three parts

- ☐ Header information
- ☐ Variable declarations
- ☐ Trailer information

Since COMGEN works as a finite state machine, the data in each section may occur anywhere in the file. The header information is copied, without modification, into the top of the Fortran insert file. The declarations for variables are copied into both the Fortran and C/C++ files with the appropriate mappings applied for variable names etc. The trailer information is copied, without modification onto the end of the Fortran insert file.

Changing the process mode/state

It is possible to have more than one occurrence of each section within a single source file. This is achieved by using the directives `.TOP`, `.VARIABLES` and `.BOTTOM`. These directives should appear with the full stop in column 1 and should be the only entry on the line. The initial state for COMGEN is to accept variable declarations.

INCLUDE directive

It is possible to have declarations spread over several files by using the `#INCLUDE` directive. This must be the only statement on a line with the `#` appearing in column one. The remainder of the line should contain the path name of the file to be processed next. Here are two examples.

```
#INCLUDE graphics
#include c:\common\errors.src
```

At the end of each include file, processing will continue with the line following the include directive. Include files may be nested (see the limitations listed on page 221).

Comments

Comments may appear either as a full line or as a partial line comment. A comment is started with `/*` and continues to the end of the line. The following examples all include valid comments:

```
/*
/* Include directives
/*

#include c:\common\colours.src  /* Colour definitions.
```

Variable declarations

Several variable declaration sections may appear in any single file. They may be interspersed with header and trailer sections at any point in the file. By allowing this, you can define two variables and use the **.BOTTOM** directive to place the equivalence statements at the end of the file.

A variable declaration has the following format:

name storage_type data_type [value] [comment]

Only the first three fields are compulsory for all entries.

name is the name of the variable to be declared.

storage_type refers to the linkage properties of the name. This may be one of **PARAMETER**, **EXTERNAL**, or **INTRINSIC**. In the case of a common block name, the name should start and end with an oblique “/” character.

data_type gives the Fortran data type of the name. This entry may be any valid Fortran data type given in table 17-1.

value is only relevant to names with the *data_type* **PARAMETER**. This field gives the actual value for the name.

comment may be used to give further information about *name* and its use.

Example

The following example file illustrates the format of the **COMGEN** source file together with the generated insert files.

COMGEN source file:

```
/*
/* Source file for a file control block data structure.
.TOP
C
C   Common block declarations for FILE data structure.
C
.VARIABLES
FILENAME           /FILE/                CHARACTER*128
POSITION           /FILE/                INTEGER*4
ACCESS_MODE        /FILE/                INTEGER*4
HANDLE             /FILE/                INTEGER*2
/*
/*  Values of the access mode flags.
/*
```

READING	PARAMETER	INTEGER*4	1
WRITING	PARAMETER	INTEGER*4	2

Issuing the command:

```
COMGEN file.src file.ins file.h /nt
```

results in the **FILE.INS** and **FILE.H** containing the following:

FILE.INS

```
C      File generated from C:\tmp\file.src
C
C      Common block declarations for FILE data structure.
C
      CHARACTER*128 FILENAME
      INTEGER*2 HANDLE
      INTEGER*4 READING,ACCESS_MODE,POSITION,WRITING
      PARAMETER(READING=1,WRITING=2)
      COMMON/FILE/ FILENAME,POSITION,ACCESS_MODE,HANDLE
```

FILE.H

```
#define reading 1
#define writing 2
struct _x1{
    char _x2[128];
#define filename FILE_._x2
    int _x3;
#define position FILE_._x3
    int _x4;
#define access_mode FILE_._x4
    short int _x5;
#define handle FILE_._x5
};
extern struct _x1 FILE_;
```

It is possible to access any of the variables defined in the source file directly in both Fortran and C. Note, however, that the names have been translated to lower case for the C definitions. Note also that (even though the common block is mapped onto a data structure) in C you access the variables directly by name rather than by referencing the structure and its element. The “_” character in the C file will be appended by the Fortran compiler automatically. This allows the Fortran compiler to differentiate between variable and common blocks during compilation and so needs to be explicitly added for C.

Data type mapping

The following table gives the mapping from the Fortran data types to the C data types.

Fortran	C
INTEGER*1, CHARACTER	char
INTEGER*2, LOGICAL*2	short int
INTEGER*4, LOGICAL*4	int
REAL*4	float
REAL*8	double
CHARACTER*(<i>x</i>)	char [<i>x</i>]

Table 17-1

Limitations

The following limitations apply to the source file.

Line length	160 characters
Maximum number of names	5000
Name length	40
Nesting level for include files	10
Number of common blocks	29

Calling the Windows API (Win32)

Introduction

This chapter describes how to call Win32 API routines from Fortran. Details of how to use a `C_EXTERNAL` function for this purpose under Win16 are given in the *ClearWin+ User's Guide*. This chapter provides a temporary addendum to the *ClearWin+ User's Guide* and describes how a `C_EXTERNAL` function under Win16 is replaced by a `STDCALL` function under Win32.

Calling Windows API routines from Fortran

As the Windows API is based upon C++, it is easier to use the API from a C++ program. It is possible to program the API from Fortran. However, Fortran data structures do not easily map on to the data structures that are used by the Windows API. One way forward, is to employ mixed language programming, keeping your existing Fortran as far as possible unchanged, and using C++ to provide an interface to the Windows API.

Programmers who are not familiar with C++ will probably prefer to avoid learning a new language. In which case the following points should be kept in mind when calling Windows API functions from Fortran.

Owing to the fact that the Windows API routines written in C++ are `__stdcall` functions, it is necessary to use a `FTN77 STDCALL` function rather than a `C_EXTERNAL` function to pop all the the arguments that are pushed on the stack when the function returns. Windows API routines (and `__stdcall` routines) pop all the arguments that are pushed on the stack by the routine that is called before returning. `STDCALL` statements for Windows API are included in the file *win32api.ins* in the default directory.

The syntax of the declaration for a STDCALL function is similar to C_EXTERNAL statement and is as follows:

```
STDCALL name ['alias'] [(desc , ... )] [:restype]
```

where:

name

is the name by which it will be called in the Fortran program.

alias

is the C++ Windows API name (or the required **__stdcall** function name). Note that this appears in single quotes and is *case-sensitive*.

desc

is an argument descriptor, and is either REF, VAL, STRING, INSTRING, or OUTSTRING (WINREF, WINSTRING, etc.. which are used in Windows 3.1 will be interpreted as REF, INSTRING, etc).

STRING, INSTRING and OUTSTRING may be followed optionally by an integer in parentheses. This integer specifies the maximum length for the corresponding argument in the C routine, in each case where the length of the corresponding Fortran character object cannot be determined (i.e. the actual argument is CHARACTER*(*)). If the integer is not specified, then a default value of 256 (bytes) is assumed for the maximum length of the string.

restype

is the type of the function. If this does not appear then the function does not return a result (equivalent to the C type void). Valid types are INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL and STRING. INTEGER, REAL and LOGICAL may be followed by a length specifier of the form "**n*". If the function result is declared to be of type STRING then the function result should be assigned to a variable of type CHARACTER.

Some examples of valid STDCALL declarations are:

```
STDCALL SUB1, SUB2
STDCALL SUB2 'GetAttr':INTEGER
STCALLL CSUB3(REF,STRING(20)):STRING
STDCALL SUB4 'GetSize' (REF,VAL,VAL,INSTRING,OUTSTRING(100))
```

If no argument specifiers are specified, then default argument linkage is assumed. This is as follows:

Arrays:

by reference (i.e. as a pointer)

INTEGER and REAL scalars:

by value

LOGICAL:

by value, as an integer of the appropriate length, 1 representing .TRUE. and 0 representing .FALSE.

CHARACTER objects:

Copied to a compiler defined temporary variable. A trailing null is added to the end of the *significant* length of the string (i.e. there are no trailing spaces). The temporary variable is then passed by reference. In addition, if the actual argument is a scalar or array element, the result in the temporary variable is copied back, and padded to the right with blanks if necessary. This is equivalent to the **STRING** linkage descriptor described below.

Where linkage descriptors are specified, the number of arguments in each call must agree with the number of descriptors specified. The various categories of objects which may correspond to particular argument descriptors are as follows:

Numeric and LOGICAL scalars:

Value or reference (VAL or REF)

Arrays, externals, dummy procedures:

Reference (REF)

CHARACTER objects:

Reference or string (REF, STRING, INSTRING or OUTSTRING)

The three variants of the **STRING** descriptor are as follows:

STRING

The corresponding argument is both input and output, and is copied to a temporary variable on entry to the routine (with a trailing null inserted at the end of the significant length), and if the argument is a scalar or array element, is copied back to the actual argument, blank padded to the right if necessary.

INSTRING

The corresponding argument is an input argument with respect to the external routine. The argument is only copied to the temporary variable, and not copied back.

OUTSTRING

The argument is returned by the external routine. The temporary variable is set up to be the length of the corresponding scalar or array element plus one, or of specified or default (256) length if the corresponding argument is **CHARACTER***(*****), but the value is only copied out. Obviously, this descriptor is only appropriate where the actual argument is a scalar or array element.

workspace, but the string must be **NULL** terminated beforehand and blank-padded on return.

When it is required to pass a **NULL** pointer to a string, the value 0 (zero) should be used.

Some Windows API functions allow a particular argument to take two different types in different circumstances. For example, an LPSTR in some circumstances and an integer in others. This is outside the scope of the STDCALL mechanism. If this feature affects you then you should copy the STDCALL statement for the relevant API and modify it to have a different Fortran name and argument list, but keeping the same called name.

A common example is the Windows API function **LoadCursor** which is used to load either a cursor defined in the program resource or a predefined system cursor. This has the following definition:

```
HCURSOR WINAPI LoadCursorA(HANDLE hInstance, LPSTR lpCursorName)
```

When used to load a cursor from the program resource, *hInstance* is the instance handle of the application. *lpCursorName* is a character string containing the name of the cursor in the program resource. This form of the function will have the STDCALL declaration :

```
STDCALL LOADCURSOR 'LoadCursorA' (VAL, INSTRING):INTEGER*4
```

When used to load a predefined system cursor, the first argument *hInstance* is set to zero and the second argument *lpCursorName* is an integer containing one of a number of predefined values which specifies the cursor to be loaded. This form of the function will have the STDCALL declaration :

```
STDCALL LOADCURSOR 'LoadCursorA' (VAL, VAL):INTEGER*4
```

However, having two differing STDCALL statements for the same Fortran function is not allowed. The solution is to change the Fortran name. For example

```
STDCALL LOADCURSOR1 'LoadCursorA' (VAL, INSTRING):INTEGER*4
```

```
STDCALL LOADCURSOR2 'LoadCursorA' (VAL, VAL):INTEGER*4
```

The same situation arises with some other functions that have the form as **LoadCursor**, for example **LoadBitmap** and **LoadIcon**,

A further example is given by the Windows API printer **Escape** function which can take many different forms. One of these has the following form:

```
int WINAPI Escape(hdc, GETTECHNOLOGY, NULL, NULL, lpTechnology)
```

lpTechnology is an LPSTR (long pointers to strings) so the STDCALL declaration for this form of the function for use in a Fortran program would be:

```
STDCALL ESCAPE 'Escape' (VAL, VAL, VAL, INSTRING,  
+ OUTSTRING):INTEGER*4
```

A second form of this **Escape** function is:

```
int Escape(hdc, SETCOPYCOUNT, sizeof(int),  
          lpNumCopies, lpActualCopies)
```

lpNumCopies, and *lpActualCopies* are both LPINT (long pointers to integers) so the **STDCALL** declaration for this form of the function would be:

```
STDCALL ESCAPE 'Escape' (VAL, VAL, VAL, REF, REF):INTEGER*4
```

As before, having two differing **STDCALL** statements for the same Fortran function is not allowed and the solution is to change the Fortran name. For example

```
STDCALL ESCAPE1 'Escape' (VAL, VAL, VAL, INSTRING,  
+ OUTSTRING):INTEGER*4  
STDCALL ESCAPE2 'Escape' (VAL, VAL, VAL, REF, REF):INTEGER*4
```


Using LINK77, RUN77 and Libraries (DOS/Win16)

Introduction

Although the load-and-go mechanism is easy to use, it is not usually suitable for finished programs. When a program is ready for use, it should be compiled and then linked by using either :

- the LINK77 utility (described in this chapter) or
- the /LINK compiler option (see page 40).

The LINK77 utility can also be used for mixed language programming to link Salford C++, FTN90, or Sheffield Pascal modules with FTN77 modules.

FTN77 produces relocatable binary (RLB) code which is *not* loadable with the standard Microsoft LINK utilities. This is because LINK works with real mode addresses while FTN77 requires a 32-bit link. The utility LINK77 is used to produce .EXE files which execute by invoking DBOS to switch into 32-bit protected mode.

This chapter gives details of the LINK77 utility and also describes two other utilities: RUN77 and MKLIB77. The usage of these utilities can be summarised as follows:

- Use LINK77 to link RLB modules in object files produced by FTN77 etc. into execute files which usually will be run directly.
- Use RUN77 to run an execute file in certain special circumstances.
- Use MKLIB77 to combine RLB modules from object files into an RLB library (or just a new combination of modules) for use with LINK77. The essential difference between such a library and a simple list of modules is that a library is scanned selectively by LINK77 in order to satisfy calls to any missing routines.

- Use LINK77 to create a dynamic link library (DLL). By default a DLL is given the .LIB extension.

The LINK77 utility

LINK77 reads a set of commands from a file or from the keyboard. If LINK77 is invoked with a file name argument then this is used as the source of commands, otherwise commands are read from the keyboard. In the latter case, LINK77 prompts for commands with a '\$' sign. When reading commands from a file all errors are fatal, but when operating interactively LINK77 recovers from errors whenever possible.

LINK77 commands

The following commands are available.

1. *Loading and Saving*

LOAD <pathname> *or* **LO** <pathname>

This command loads RLB modules from a file produced by FTN77, Salford C++ or Sheffield Pascal. The .OBJ suffix is added to the file name if necessary. This command supports the use of DOS wildcards (e.g. LO *.OBJ). Object files and RLB libraries produced by MKLIB77 can also be loaded with this command but modules from RLB libraries are only loaded in order to satisfy calls to missing routines.

LOAD_EXHAUSTIVE <pathname> *or* **LE** <pathname>

This command is used to load an RLB library and in this context it is identical to LOAD, except that the library file is scanned repeatedly until no more unresolved references can be satisfied. This means that it is not necessary to order the routines in the library file. The repeated scanning is performed on a dictionary stored at the front of the file, so no inefficiency is implied by this process. Use an explicit filename extension with this command.

FORCELOAD

This command is also used to load an RLB library and is equivalent to LOAD in the sense that the whole of the library is loaded without selectivity.

FILE <pathname>

This completes the link process and puts the result in the specified file (appending the .EXE extension if necessary). If the pathname is omitted then LINK77 uses the first .OBJ filename to be loaded as the root for the .EXE pathname. (When LINK77 is used to produce a DLL, see page 237, the .LIB extension is appended.)

If LINK77 finds any unresolved references, it checks any currently available dynamic link libraries to see if the reference can be resolved. If not it reports the function(s) as missing. Regardless of whether it finds the function in question, LINK77 sets up calls to the function so that they will snap a dynamic link if executed. Note that this means that you can link a program in the absence of the dynamic link libraries that it calls upon.

LARGE_FILE

Normally LINK77 produces an executable file which must be loaded in its entirety by DOS (with the usual real mode memory constraints) even though the program will run in protected mode and access additional memory. RUN77 can be used if a .EXE file becomes too large to load in DOS. However, if the LARGE_FILE command has been given, then the FILE command produces a file which requires much less DOS memory. DBOS then reads the file into memory again. This incurs a slightly larger startup cost but enables very large executables to be loaded from the DOS command line without running RUN77. To be effective, the main program should be loaded as early as possible when using this option.

PERMIT_DUPLICATES

After this command has been given the linker will not abort when a function or subroutine is loaded more than once. Instead a warning message will be given and all references to the routine will use the version of the routine that was first loaded. This is sometimes useful to effectively replace a function or subroutine in a large file of relocatable binary.

INCLUDE <pathname>

This command executes the linker commands from the given file. Command files can be nested to a level of 10 deep. If you enter this command from the keyboard then the included commands will be executed and control will be returned to the keyboard. If the command is called from within another INCLUDE file then control will be returned to the line after the command.

QUIT *or* Q

Terminates the linking process without saving anything on the disk.

2. Diagnostic Information

MAP <pathname>

This writes a map of the load to the file <pathname>. No suffix is implied. If <pathname> is omitted the output is directed to the screen.

XREF <file>

Specifies that a cross reference map should be written to <file>. This command must be issued before any code is loaded.

NOTIFY <name>

Tells LINK77 to report whenever it encounters the given name. A typical use for

NOTIFY would be to determine exactly where an unresolved external reference occurs (as identified by a previous run of LINK77).

REPORT_DEBUG_FILES

After this command the linker will keep a list of all loaded files that contain debugging information (i.e. compiled with /DEBUG, /CHECK, /FULLCHECK or /UNDEF). After linking, LINK77 will produce a list of all the files it has encountered with debugging information.

SUPPRESS_COMMON_WARNINGS

This will suppress the warning messages issued by LINK77 if a common block is defined with different lengths in different routines. However, if you initialise a common block and subsequently redefine it to be of greater length the result will still be a fatal error.

*

Lines beginning with an asterisk are treated as comments and are ignored by LINK77.

3. Dynamic Link Libraries

LINK77 can also be used to create dynamic link libraries. The following summary lists the commands which relate specifically to DLLs. Further details appear on page 238.

LIBOFFSET <hex number>

When LINK77 is used to create a DLL, the command sequence begins with LIBOFFSET with <hex number> being the address at which code in the library will be designed to run.

SUPPRESS

Specifies that subsequent routines loaded into a DLL will not be callable from outside that library.

NOSUPPRESS

Cancels a previous SUPPRESS command.

ENTRY <routine name>

Specifies that the function is callable from outside the library regardless of the state of the suppress flag. Suppose that you have a DLL consisting of many functions of which only one is meant to be called from outside the library. A simple way to enforce this is to load the code in SUPPRESS mode and then specify the name of the function in an ENTRY command.

PRESERVE_CASE

This command tells LINK77 not to convert the names of symbols in commands to upper case. For example, after the use of this command it is possible to set the address of a lower case symbol using the SY command. This command is normally used in conjunction with the C compiler.

4. Common Blocks and DLLs

SYMBOL <common block name> <hexadecimal address>

This command (which can be abbreviated to **SY**) defines the start address of a common block (which must not be initialised in a block data routine). This is of most use in conjunction with DLLs to enable a library to share a common block with a program or other DLL. If you use this command you should ensure that the storage you specify does not overlap with anything else.

COMMON_BASE <hexadecimal address>

The **COMMON_BASE** command tells **LINK77** where to start allocating common. This command is used in conjunction with the **DEFCOM** command.

DEFCOM <common block name>

This command forces a common block to be allocated at once. Usually it is used in conjunction with the **COMMON_BASE** command, and this is described in more detail in conjunction with DLLs (see page 239).

Using LINK77

As an example of the use of **LINK77**, consider linking the **MYPROG.OBJ** and **SUBS.OBJ** files to produce a run file **MYPROG.EXE**. The following commands could be used:

```
LINK77
LOAD MYPROG
LOAD SUBS
FILE
```

LINK77 can also be used to create dynamic link libraries as described on page 238.

Running the program

The resulting **.EXE** file can be executed by typing its name - in the same way as for any other **.EXE** file. Any command line parameters can be read with **CMNAM@** (see the *FTN77 Library Reference* manual or the on-line Help system). If the program fails, the result will be a brief diagnostic and a register dump. These dumps are mainly useful in conjunction with assembler coding (see chapter 15). In order to run the program in this way the **.EXE** file must be small enough to be loaded by DOS (if not, DOS will give the error "Program too big for available memory"). Since space for uninitialised common and dynamic variables is not reserved in the file, most programs will be loadable in this way. However, programs which contain really large amounts of code may be too large and must then be loaded with the **RUN77** utility.

The RUN77 utility

RUN77 is useful for two reasons:

- In order to load and run .EXE files which are too large to load directly under DOS. (An alternative solution to this problem is use LINK77 with the LARGE_FILE command.)
- In order to make the facilities of the interactive debugger available to pre-linked programs.

The command is used as follows:

```
RUN77 <pathname> options
```

The .EXE suffix is automatically added to the pathname if it is not present. The program is run as normal except that control is passed to the interactive debugger in the event of a run time fault.

The following options are available:

/BREAK

This causes the program to be suspended in the interactive debugger in the same way as the /BREAK option operating with FTN77. The relevant routines must have been compiled with the /CHECK, /FULLCHECK, /UNDEF or /DEBUG options for this to be useful.

/HARDFAIL

This suppresses the interactive debugger intervention in the event of a run time fault. This option is useful if RUN77 is simply being used to load a program which is too big to fit into DOS memory.

/PARAMS

The remainder of the command line after /PARAMS is left for the program to read. For example, if the program reads its input and output files from the command line (using CMNAM@, see the *FTN77 Library Reference* manual or the on-line Help system) the command line might look as follows:

```
RUN77 MYPROG /PARAMS INFILE OUTFILE
```

/PRELOAD

Using this option forces the whole executable file to be loaded before execution begins, rather than being paged in on demand. This can be used to remove the effect of the progressive paging in of a program on any timings which might be taking place, and also to check whether the run file will fit in its entirety into a given memory size.

/UNDERFLOW

This causes floating point underflow to be treated as an error. By default, calculations which underflow produce a zero result.

`/READ <unit> <pathname>`

The `/READ` option opens the given file for formatted sequential read access on the given unit. `/WRITE` is similarly used to assign an output file from the command line. `/READU` and `/WRITEU` are correspondingly used for unformatted sequential access files. For example:

```
RUN77 MYPROG /READ 7 MYPROG.DAT
```

`/READ` and `/WRITE` etc. can be used in one and the same command line and may be used more than once in the same command line.

Note:

`RUN77` passes its arguments to the system routine `START_PROGRAM@` (see the *FTN77 Library Reference* manual or the on-line Help system), which provides equivalent facilities under program control.

Libraries

The `FTN77` system supports two kinds of libraries:

- ☐ Relocatable binary libraries for use with `LINK77`.
- ☐ Dynamic link libraries. (These are discussed on page 237.)

Relocatable binary libraries

RLB libraries are prepared using the `MKLIB77` utility. This utility has two modes - *interactive* mode and *command* mode.

RLB libraries are scanned by the `LINK77 LOAD` command and subroutines which satisfy currently outstanding references are loaded.

Conceptually this process is linear. For example, suppose that `FUNC1` contains a reference to `FUNC2` and assume that neither of these routines have been loaded. If the routines were to appear in the library in the order

```
FUNC1
FUNC2
```

then both would be loaded with the `LOAD` command. If, however, the order was reversed, `FUNC2` would not be loaded unless an explicit reference to `FUNC2` was outstanding. This mechanism can be used to achieve special effects within libraries.

Sometimes it is inconvenient or impossible to order an RLB library in an appropriate manner. In this case the LINK77 command LOAD_EXHAUSTIVE should be issued instead of LOAD.

1. MKLIB77 command mode

In order to prepare a relocatable binary library you must first produce a file containing the relocatable binary corresponding to the routines in question. In the simplest situation, these routines will already reside in one file and can be compiled by a simple call to FTN77. However, if the project involves a number of files, the wildcard form for the file name can often be used together with the /BINARY compiler option (see page 22). Alternatively after compilation, .OBJ files may be combined if necessary using the DOS COPY command. For example:

```
COPY F1.OBJ/B+F2.OBJ TEMP.OBJ
```

where /B is used to inform COPY that the files are binary files. If a routine in the library references another, then that routine must precede the referenced routine in the .OBJ file. In the above example, routines in F1.OBJ may call routines in F2.OBJ, but not vice-versa. The file is converted into a library using the MKLIB77 utility thus:

```
MKLIB77 <relocatable binary file> <library file>
```

For example, continuing the above example:

```
MKLIB77 TEMP.OBJ MYLIB.LIB
```

2. MKLIB77 interactive mode

Interactive mode is entered by typing the MKLIB77 command with no arguments:

```
MKLIB77
```

If there are no .OBJ files in the current directory the user is prompted for another directory. When a directory is reached which does contain .OBJ files, these are presented in a menu on the right of the screen, and the user selects one of them by moving the cursor bar to the file in question, and pressing **Enter**.

Once a relocatable binary has been loaded into the system, the routine names (and any entry points) will be scrollable in the window on the left of the screen.

The following keys can then be used in addition to the normal cursor keys:

Del	Deletes the routine at the cursor
Ins	Prompts for another .OBJ file to be inserted above the routine at the cursor. Position the cursor just beyond the last routine in order to append relocatable binary to the end. A window of relocatable binary files is displayed for you to select the one to be inserted

Alt-L	Mark the beginning or end of a block of routines. The block will be marked in red
Alt-D	Delete a block of routines
Alt-M	Move a block of routines to just above the current cursor position (which must not itself be in the block)
Alt-C	Copy a block of routines to just above the current cursor position (which must not itself be in the block)
F3	Create a new file and exit. You will be asked for the name of the file and whether the result is to be an RLB library or simply an object file (include an explicit extension, .OBJ could be used for both types of file).
Alt-S	Prompts for the name of a routine or entry point and searches for the routine in the currently loaded relocatable binary
Esc	Exits the utility

In either the command or the interactive mode, MKLIB77 will reject a .OBJ file which contains a main program. The result can be used with LINK77 using the LOAD command in exactly the same way as any other relocatable binary. Only those routines that are required will actually be loaded by LINK77.

Dynamic link libraries

Dynamic link libraries are the preferred method for delivering large packages of routines. These libraries operate in much the same way as the DBOS system library, in that the code which they contain is linked into the program as it is required at run time. This means that .EXE files can be kept small and link times are fast. When DBOS is invoked it looks for the file LIBRARIES.DIR (actually LIBRARIE.DIR since DOS file names are limited to 8 characters) in the directory containing the DBOS system. This file, if present, should consist of a list of pathnames (not local names) of the dynamic libraries to be used subsequently. Up to twenty such libraries can be specified. Each pathname should be on a separate line in the file. For example, a typical LIBRARIES.DIR file might contain:

```
C:\SYSLIB\GKSLIB.LIB
C:\MYLIBS\MATRIX.LIB
```

A dynamic link library is a piece of absolute binary code which has been linked so as to work from high numbered addresses. This code is placed in a .LIB file together with some map information which enables DBOS to load the code as it is actually needed at run time. Note that the fact that the library code is loaded at high addresses does not imply that a correspondingly large amount of physical memory must be available on your PC. This is because DBOS uses the virtual memory hardware on the 32-bit Intel chip.

The address at which a library is loaded is arbitrary, provided it is at least 4095 bytes beyond the last address used by a program, and not greater than 60000000 (to avoid space used by DBOS). It is suggested that start addresses (hexadecimal) of 41000000, 42000000, 43000000, etc. are used, as these are well clear of any loaded programs and the stack. If more than one library is in use, *their address spaces must not overlap*. The above addresses provide for 16 megabytes of virtual address space per library. Details of the Salford DBOS memory map are given on page 318.

Creating dynamic link libraries

Dynamic link libraries are created using LINK77. The first command to this utility must be:

```
LIBOFFSET <hex number>
```

The hexadecimal number is the address at which code in the library will be designed to run. The remaining commands are the same as those used to create a .EXE file except that you must not load a main program into a library file. A file with a .LIB suffix should be used on the FILE command. As an example, the following commands could be used to convert a set of Fortran routines in MYLIB.FOR into a dynamic link library called MYLIB.LIB:

```
FTN77 MYLIB

LINK77
LIBOFFSET 41000000
LOAD MYLIB
FILE MYLIB.LIB
```

A subroutine call will only reference a dynamic link library if there is no routine of the same name to be found in the program itself. This is useful because it means that the names of internal routines in a library will not clash with user-defined routines in the way which they would if the library of routines were directly linked into the program. It is also possible to hide the internal routines of a library completely. This is done using the SUPPRESS and ENTRY commands in the linker. For example, consider that in the above example only functions GRAPH1 and GRAPH2 are to be callable by the user.

The library could be loaded thus:

```
FTN77 MYLIB

LINK77
LIBOFFSET 41000000
SUPPRESS
LOAD MYLIB
ENTRY GRAPH1
```



```
ENTRY GRAPH2  
FILE MYLIB.LIB
```

Sometimes a dynamic link library may contain a call to a routine which is located in the user's program. For example, an integration routine may make calls to a user-supplied routine called **FUNC** (say) to supply function values. If the call is made by means of an external routine passed into the library as an external then there is no problem. If the name is hard coded in the routine and the routine is not found within the library, all other libraries plus the system library are searched for the routine. If the routine reference is still unsatisfied, the user's program is searched for the reference. This means that references from within a library to user-supplied routines will work provided a routine of the same name does not exist elsewhere in the system.

Common blocks in dynamic link libraries

Special consideration needs to be given to common blocks which are used to communicate information with a dynamic link library. A library may use as many common blocks as required internally to itself. However, if a program contains a common block **/C/** (say) and calls a routine in a library which also references a common block **/C/**, then these two common blocks will not, by default, be the same. This is because each piece of code, having been linked quite separately, will reside at different addresses.

Sometimes this can be quite useful, however to share a common block you should specify its address using the **LINK77** command **SYMBOL**. If the same address is used when loading the program and the library then all will be well. In general it is a good idea to specify an address which is well removed from the program and the library. This procedure can become tedious in situations in which there are large numbers of named common blocks, since (if common blocks are allocated using the **SYMBOL** command) it is the user's responsibility to ensure that blocks do not overlap.

The **LINK77** commands **COMMON_BASE** and **DEFCOM** provide a means of overcoming this problem. By means of these two commands, common blocks may be loaded starting from a given address, without the need to calculate the positions of subsequent blocks. For example if you had a common block **/A/** of length 1000 bytes (hex 3E8 bytes) and another common block **/B/** of length 500 bytes, then the commands:

```
COMMON_BASE 30000000  
DEFCOM A  
DEFCOM B
```

would load common block **/A/** at address 30000000 (hex) and block **/B/** at address 300003E8. (Note that common block sizes are rounded up to the nearest multiple of 4 bytes for the purposes of allocation for hardware efficiency reasons.) These commands would have to follow the load of relocatable binary which referenced the two common blocks (in order that **LINK77** could determine their sizes). If these

commands are included in the link of the library and of the program, then common blocks /A/ and /B/ will be shared.

If a common block has been initialised in a **BLOCK DATA** statement there is no easy way to share it between a program and a library.

SLINK (Win32)

For information about the Salford DOS/Win16 linker LINK77 see chapter 19.

Introduction

SLINK is Salford Software's 32-bit linker for Win32. It is designed to accept Win32 COFF object files and produce Win32 libraries (.LIBs), Win32 Portable Executable (PE) executables (.EXEs) and Dynamic Link Libraries (.DLLs). SLINK has been designed to make it powerful and easy to use.

SLINK will act either as a library builder or as a conventional linker or both simultaneously. SLINK is tailored for object code produced by Salford compilers. It can, however, be used with COFF object code produced by other compilers. SLINK will not accept 32 bit OMF object code, the native object code format for OS2/2.

Getting started

SLINK has three modes of operation:

- a) command line mode,
- b) interactive mode and
- c) script file mode.

Command line mode takes all parameters from the command line whilst interactive mode processes commands one at a time as they are entered from the keyboard. This is very similar to LINK77, the Salford linker for the DBOS family of compilers. Script file mode reads the commands from a text file. This has two variations, a Salford LINK77 compatible command mode and a Microsoft compatible command mode.

It is easy to build executables with **SLINK**. For example, suppose that you compiled a program contained within one file, say **MYPROG**. The compiler will produce an object file called **MYPROG.OBJ**. To produce an executable from this, the following command line will suffice

```
slink myprog.obj
```

In response, **SLINK** will:

- 1) Load **MYPROG.OBJ**.
- 2) Set the default entry point for Salford programs.
- 3) Scan the Fortran library, **FTN77.DLL** or **FTN95.DLL**.
- 4) Scan the Salford C library, **SALFLIBC.LIB**.
- 5) Scan the default list of system DLL's.
- 6) Set the file name to **MYPROG.EXE** (derived from the name of the object file).
- 7) Create the executable.

This command line illustrates **SLINK**'s command line mode. Alternatively, we could use **SLINK**'s interactive mode in the form:

```
slink
$ load myprog
$ file
```

Note that **SLINK**'s command prompt is a \$, and that **SLINK** has provided the **.OBJ** extension. Interactive mode always terminates with a **file** command. The **file** command is used both to terminate the session and to optionally provide the filename that is to be used to store the output. **SLINK** will know that you are building an executable and automatically supplies the **.EXE** extension.

Command line mode

This is an example of how to use **SLINK** in command line mode:

```
slink myprog.obj -file:test
```

In command line mode, all of **SLINK**'s commands begin with “-” or “/”. Any parameters are separated from the command by a colon “:”. Note that there must be no spaces within the command (in this case **file:test**). Where the command does not take parameters, it should not be terminated with a colon. Where parameters are optional because **SLINK** will complete the command (for example the **file** command) then the colon is also optional.

Linking multiple object files

Multiple object files can be linked:

- 1) in command line mode by placing more objects on the command line,
- 2) in interactive mode by using more **load** commands, and
- 3) in script file mode by modifying the script file in a manner corresponding to 1) for command line mode or 2) for interactive mode.

Abbreviating commands

Many of **SLINK**'s commands have an abbreviation. These are shown in the command reference (see the end of this chapter). For example, instead of the **load** command you may use **lo**. Also, many of **SLINK**'s commands have an alias.

Script or command files

When large numbers of commands are needed or the same command sequence is repeated many times it is helpful to place the commands in a script or command file. Interactive mode script file names are prefixed with a "\$" on the **SLINK** command line, whereas command line mode script files are prefixed with an "@". Commands taken from script files are presented in the same form as that used when entering commands from a command prompt in interactive mode or from the command line. For example,

```
slink $myprog.inf
```

will tell **SLINK** to take its commands from a file called **MYPROG.INF** and that the command format is interactive mode, whilst

```
slink @myprog.lnk
```

will tell **SLINK** to take its commands from a file called **MYPROG.LNK** and that the command format is command line mode.

Note that the file suffixes **.INF** and **.LNK** are purely conventional and do not affect how the commands will be interpreted - you may use suffixes of your own choosing if you wish.

As a special case, for interactive mode command files, the \$ before the file name can be omitted. In this case, if the file is not recognised as a **COFF** object, it will be opened as a script file. For this reason, **COFF** objects specified on the command line must have the correct filename extension as **SLINK** will not complete the filename itself.

More than one script file may be specified on the command line but script files may not themselves contain script files.

Differences between command line mode and interactive mode

The main difference between command line mode and interactive mode is that command line commands (i.e. commands that begin with a “-”) are implemented first and objects and libraries are loaded later. Commands have a deferred effect and can appear anywhere in the command line or script file and in any order. For example,

```
slink myprog.obj -file:test
```

and

```
slink -file:test myprog.obj
```

have exactly the same effect. In the latter case, specifying `-file:test` first, tells **SLINK** that the filename will be **TEST.EXE** but no immediate action is taken on the **file** command.

Interactive mode commands are implemented immediately, where appropriate. For example, placing the following commands in a script file:

```
lo myprog  
file test
```

and

```
file test  
lo myprog
```

will have different effects.

The first script will do as expected, load an object file called **MYPROG.OBJ** and produce an executable from it called **TEST.EXE**.

The second script will terminate with an error since the **SLINK** session is always terminated in interactive mode by **file** and at that point no object files have been loaded.

Comments

In script files (for both interactive and command line mode) all text following the semicolon character “;” is ignored until a newline character is encountered. This makes it easy to temporarily “comment out” commands. For example in

```
slink file1.obj file2.obj ;file3.obj tpgraph.lib
```

the objects **FILE3.OBJ** and **TPGRAPH.LIB** will not be loaded.

Mixing command line script files and interactive mode script files

It is not advisable to mix interactive mode and command line mode script files due to the differences in the way that they are interpreted.

Executables

The previous section described briefly how to generate executables. This section looks at additional commands that are either useful during the production of the executable or affect the way the executable is produced.

Link map

The link map is used to examine the structure of the executable or DLL in detail. The map will show:

- 1) The entry point (see below) and its address.
- 2) All of the routines that **SLINK** could not find a definition for. These are called unresolved externals (see below). The **SLINK** map will also show the path name of the file that contained the initial reference to the symbol.
- 3) The map then lists all of the defined symbol names and their addresses together with the path name of the file that contained the definition of the symbol. These addresses show the “preferred address”. The actual run time address may be different.
- 4) The link map finally contains a brief outline of the executable by showing the addresses where the executable’s sections have been loaded.

For example, the following **SLINK** session will produce a link map named **FILE1.MAP** and an executable **FILE1.EXE**. These names are derived from the first loaded object file name.

```
slink
$ lo file1
$ lo file2
$ map
$ file
```

Unresolved externals

Unresolved externals are those symbols for which **SLINK** was unable to find a definition when searching the specified library and object files. Some omissions may be intentional and may simply be routines that will not be called. Others may be unintentional omissions. **SLINK** will successfully complete a link session even when there are unresolved externals. It will provide a temporary definition of these symbols so that, when the function is called, an error message will be printed out stating the name of the function and the address from which it has been called.

In interactive mode, the command **lure** (List UnResolved Externals) may be used at any time to check the progress of the linker session. This command will list all of the

functions for which it currently has no definition together with the path name of the file that contained the reference.

Do not be alarmed if a large number of functions are listed as unresolved when the command is used immediately before the **file** command is issued. This is quite normal because there will be functions in FTN77.DLL, FTN95.DLL, SALFLIBC.LIB (or SALFLIBC.DLL) and in the system DLLs that need to be linked. SLINK will automatically link them after the file command has been issued.

Direct linking with DLLs

SLINK allows direct linking with one or more DLLs without the need to use import libraries (see section 5). It will generate its own import library for a DLL based upon the information contained in the DLL's export table. If you produce a DLL, you have the choice whether or not to produce an import library.

Sometimes, as is the case with SALFLIBC.LIB (or SALFLIBC.DLL), the library is a combined import and standard library. In this case, the functions in the standard library part are not contained within the DLL, so directly linking with the DLL will not achieve the same result as linking with the .LIB file. This means that you should not link directly with SALFLIBC.DLL – always use the .LIB file i.e. SALFLIBC.LIB.

For example, suppose that some of the functions you need are provided inside a DLL called TPGRAPH.DLL. In this case the linker would not import the runtime code for the functions from the DLL even though the DLL must be loaded as illustrated here:

```
slink
$ lo file1
$ lo file2
$ lo tpgraph.dll
$ file
```

TPGRAPH.DLL is merely used to acquire the information that is necessary for the executable to import functions from TPGRAPH.DLL at run time.

SLINK will not search the system path for the DLL. You should specify the full path name on the **load** command.

Additional Commands

Various commands are used to provide information that is required to generate an executable. SLINK will take a sensible default for all of these commands and it is unlikely that you will need to use them.

Runtime tracebacks

SLINK builds an internal map into each executable. The location of this map is registered with SALFLIBC at runtime. It contains the true fixed-up runtime

addresses. In the event of a fault during program execution that causes the program to abort, **SALFLIBC** will print out a traceback of the various routines called, tracing back to the user's main program.

The internal map contains the name and address of all the static and external functions in your code. You may wish (e.g. for code security reasons) to remove this map and forego the run time traceback facility. This may be achieved by using the **notrace** command.

Linking for Debug

When source files are compiled using checking or debugging options, the compiler inserts additional information into the object files produced. This information has to be organised and placed into the executable so that the Salford debugger can be used to examine source files, set break points, examine variables etc. **SLINK** will automatically insert the debugging information into the executable. However, since this increases the size of the executable by a considerable amount, you are advised to switch off the checking and debugging options before preparing production versions of your executable.

The syntax of the command is:

debug [**full** | **partial** | **none**]

This means that the word **debug** can be followed by one of the options **full**, **partial** and **none**. The **debug** command with no parameters or with the keywords **partial** or **full** will insert debug information into the executable. **partial** and **full** have the same meaning. The keyword **none** will remove the debug information. The default is **full**.

If debug information is not found in the object files, then **SLINK** will not insert any debug information into the executable. In this case, if you are using the **debug** command with **partial** or **full**, then **slink** will produce a warning.

Comment text

The syntax for the command is:

comment [**on** | **off** | "*text*"]

This means that the word **comment** can be followed by one of the options **on**, **off** and some user-supplied text in quotation marks (in this chapter, user-supplied values are shown in bold italics). It is possible to embed comment text into an executable using the **comment** command. Comments are included into the .comment section in the executable (here the word *comment* is preceded by a period/full stop). Typically copyright information and version information is included in comment text. Even if the file name is changed, text within the comment section will still identify the executable as your product.

SLINK will prepend the text with the characters “@(#)”, and will also add newline characters at both ends of the text. This makes the text easy to search for with a **grep** type utility. Comment text is also added by the compiler used and by **SLINK** in order to identify version numbers used for the build. **SLINK** will always add its own comment to the executable.

Any number of **comment** commands may be issued. Text following the **comment** command should be delimited with double quotation marks (“”). The **comment** command is only available in interactive mode.

It is also possible to exclude the **.comment** sections of **COFF** objects from the executable. This is useful where, for example, your application has been linked from a large number of **COFF** objects. The **.comment** section in the executable would then normally be very large. The **comment off** command will prevent the inclusion of these comments from the point at which the command was issued until a **comment on** command is issued. User comments will still be included.

Here is an example of the use of comments.

```
comment "Mars Attack v2.03,InterGalactic Software Inc."
comment off
.....
comment on
```

Virtual Common

It is possible in most languages (and in particular in Fortran and C/C++) to have uninitialised global data, for example, a common block in Fortran not initialised with a **BLOCK DATA** subprogram. Under normal linking, these are accumulated into the **.bss** section in the executable (BSS is an old IBM term meaning Block Started by Symbol). Although this section does not contribute to the size of the executable it does contribute to the size of the loaded image. The consequence of this is that the system must have the resources available to meet the size of the **.bss** section. This is unfortunate, since many applications use very large global arrays, only some of which is ever used.

If the **SLINK** command **vc** or **virtualcommon** is used at some stage during the link process, the “**.bss**” section is removed from the executable and the global data is allocated to virtual memory at runtime. The result is that pages of memory (4Kb each) are allocated from the system on demand.

Libraries

Win32 acknowledges three types of library: Standard Libraries, Import Libraries and Dynamic Link Libraries (DLLs).

Standard libraries and import libraries

These libraries contain code that is linked into the user's program by SLINK as part of the program's executable image. They are easy to build and need no special initialisation. Win32 standard and import libraries are very similar to UNIX COFF archives and for that reason are referred to as archives. Archives consist of complete object files loaded in together with various headers. These object files are referred to as members. Win32 archives usually have the filename extension .LIB.

Import Libraries

Import libraries are used by programs that wish to link with DLLs. Import libraries are not usually needed when SLINK is used because SLINK can extract the information directly from the DLL itself. However, SLINK will generate import libraries for the DLLs it creates if requested and will also accept them as input for the **load** command. Import libraries have to be generated for other linkers that cannot extract information directly from the DLL.

Salford run time library

All programs that are compiled using a Salford compiler must be linked with SALFLIBC.LIB. SALFLIBC.LIB is a special kind of library and is a combined standard and import library. SLINK will automatically link with SALFLIBC.LIB for you. Although SALFLIBC.DLL exists, it should not be scanned directly since SALFLIBC.LIB is more than just an import library. Not all references that are satisfied by scanning SALFLIBC.LIB can be satisfied by scanning SALFLIBC.DLL.

Any import library or DLL may only be scanned once. A situation like the following is to be avoided:

```
l o object1.obj      creates references to KERNEL32.DLL
l o kernel32.dll     satisfies current references to KERNEL32.DLL
l o object2.obj      creates more references to KERNEL32.DLL
```

SLINK will automatically scan the system DLLs, in order to satisfy references in the user program, if any unresolved references exist at the end of the link process.

The following order of linking should be observed:

- 1) Object files

- 2) Non system DLLs, non system import libraries and other standard libraries
- 3) FTN77.DLL or FTN95.DLL
- 4) SALFLIBC.LIB
- 5) System DLLs

The list of DLLs included in “system DLLs” is given in reference section beginning on page 253.

Note that the last three stages are automatic but should none the less be regarded as having taken place.

Dynamic Link Libraries

Dynamic Link Libraries are special kinds of libraries used by modern operating systems. They do not contain code that is directly linkable with the user's program. They are pre-linked bodies of code that are called at run time and are a kind of executable, rather than a kind of archive. The advantage is that, when the DLL is updated, the user's program does not have to be relinked unless the order of the routines contained within the DLL has changed. Also, by using a DLL, very little code is added to the user program.

Win32 DLLs require that programs wishing to use a DLL must link with an import library (see above). This is so that the system loader can make the link between the user program and the DLL when the user's program is loaded at run time. The Salford Fortran and C/C++ runtime libraries are DLLs. Usually, runtime library routines are not linked into the user program. The exception is the case where the compiler has inserted the code inline. Thus executables that use DLLs are much smaller than would otherwise be the case.

Generation of archives

The linker command **archive** will specify that an archive is to be generated. The **archive** command is available in both command line and interactive mode.

Object files to be placed into the archive are specified using the **addobj** command. This informs the linker that the object specified is not to be included in the normal link process but is to be placed in the archive. Archives themselves may be added to the archive. In this way, objects may be added to already existing archives. You can also give the **addobj** command a *listfile* name preceded by @ that contains a list of files that you wish to be included.

The following example constructs an archive named NEWLIB.LIB which contains the object files FILE1.OBJ, FILE2.OBJ together with all the object files contained within LIBFILE.LIB. This results in two more object files being added to LIBFILE.LIB. Note how the **file** command is used to terminate the linker session and initiate building the archive.

```
slink
$ archive newlib.lib
$ addobj file1.obj
$ addobj file2.obj
$ addobj libfile.lib
$ file
```

or

```
slink
$ archive newlib.lib
$ addobj @listfile
$ file
```

where *listfile* contains the following text lines

```
file1.obj
file2.obj
libfile.lib
```

the command line form of this command would be:

```
slink -archive:newlib.lib -addobj:file1.obj
      -addobj:file2.obj -addobj:libfile.lib
```

or

```
slink -archive:newlib.lib -addobj:@listfile
```

Note that the **load** and **addobj** commands may be used with wildcards. For example,

```
addobj *.obj
```

Generation of DLLs and exporting of functions

Since DLLs are run time libraries, it follows that they can also provide routines for other applications that are running. These routines have to be exported in order to make them available to other applications. A DLL must have some exports.

The **export** command will make a function (or a variable) available to other applications by inserting it into the export table in the DLL.

If you wish all of your functions to be exported, unless otherwise specified by the **exportx** command, then the **exportall** command will insert them all into the export table.

The **exportx** command will prevent functions from being inserted into the export table. The **export** command overrides the **exportx** command.

The **dll** command is used to specify that a DLL is to be built.

The following example will generate a DLL named MYDLL.DLL. All of the functions within MYDLL.OBJ are exported.

```
slink
$ dll
$ lo mydll.obj
$ exportall
$ file
```

Note that the filename extension .DLL is appended by SLINK.

Import libraries can be generated by using the **archive** command described above. In which case, all of the exported functions will have the necessary members added to the import library to enable them to be linked with within the DLL at runtime.

The export command

The export command has the form:

export *entryname* [=internalname] [@ordinal [noname]] [data]

Only a shortened version of the command is available in command line mode, namely:

-export:entryname [=internalname]

internalname is the name of the symbol as it appears in your program or object files. Note that, in the case of **__stdcall** functions, there is an additional “decoration” added to the end of the symbol. In general, you should not use this decoration nor the leading underscore added to the symbol name. SLINK will match the undecorated name specified with the decorated name in the loaded object files. In the case of Salford C++ decorations, the full decorated name should be specified but without the leading underscore.

entryname is the name of the symbol by which the user would call your function. SLINK will append a leading underscore and transfer any **__stdcall** decoration found for the *internalname* to the *entryname*. The name of the function will appear in the DLL export table exactly specified with this command.

In the following example, suppose your **__stdcall** function **func** exists with the full decorated name **_func@12**

```
export gloop=func
```

SLINK will match **func** with **_func@12** and also export **_func@12**. The name appearing in the export list will be **gloop** and the symbol appearing in the import library (if any) will be **_gloop@12**.

The name in the export table and the name in the calling program's import table are identical. This is so that the system loader is able to find the function in the DLL.

data is used to export a data item. You must use a pointer to the data item in your program. See the command reference section below for descriptions of ordinals and the **noname** keyword.

SLINK command reference

SLINK has two basic modes of operation: interactive and command line.

Generally, where a file name is optional the default file name is generated by taking the file name of first loaded object file and adding the appropriate extension.

Interactive mode

This mode takes commands in a similar form to LINK77, the DBOS linker. The commands are order dependent with the exception that the **map** command may be given at any point.

A list of the interactive mode commands is given below. Note that the alias is given in brackets alongside the command and that all commands are case insensitive.

addobj [*filename* | *@listfile*]

The specified COFF object is to be included in a COFF archive. Only COFF object and COFF archive files may be so loaded. PE executables and dynamic link libraries (DLLs) may not. This allows COFF archives to contain ReLocatable Binary (RLB) code and also be an IMPort LIBrary (IMPLIB) for a DLL. SALFLIBC.LIB is such an example, it is an import library for SALFLIBC.DLL and yet contains RLB for the startup procedure SALFStartup.

Alternatively, a list of COFF objects to be included may be inserted in a *listfile* the path of which is preceded by an @.

archive (implib) *filename*

Specifies that an archive is to be generated from objects loaded with the **addobj** command. It also specifies that an import library is to be generated from the export list, if it is non-empty.

comment [**on** | **off** | "*text*"]

text is inserted into the .comment section in the executable. The text must be delimited by a quotation mark ("). Alternatively you can use **on** or **off** to enable the inclusion of .comment sections from COFF objects from that point onwards in the link process.

decorate

Symbols in the map and in the listing of unresolved externals are normally

reported in their undecorated form. This command will force symbols to be reported in their decorated form.

dll (library) *modulename*

Specifies that a DLL library is to be generated. DLLs have an internal name, distinct from the filename, used by the system loader to recognise the DLL. The DLL command also sets the internal name (i.e. the module name) that the DLL is known as to the system loader to *modulename*.

If a module name is not specified, then the module name is generated from the file name with a .DLL suffix. Note that this is equivalent to the **library** keyword in a module definition file (.DEF file). The default suffix for the file command is set to .DLL.

For example, one of the system DLLs, USER.DLL, has a filename USER32.DLL. By default, SLINK will set the internal name of the DLL to be the same as the filename.

entry *symbol*

Specifies the entry point for the program. For linking Salford compilations, this command is unnecessary as the entry SALFStartup is assumed. If used, this command MUST be the first command in the SLINK session. If this command is not used, then the entry point will be set to SALFStartup after the first object file has been loaded. If an entry point other than SALFStartup has been specified, this will disable the default loading of SALFLIBC.LIB.

export *entryname*[=*internalname*] [@*ordinal* [*noname*]] [*constant*]

This has the same syntax as an entry in the **exports** section in a .DEF file. It adds an entry to the export list. The *entryname* specified does not have to exist but if it does not it may cause a run time error if the entry point is used. If the DLL command is not used the module name is generated from the file name with a .EXE suffix. Overrides the **exportx** command.

Note: Only a shortened version of this command is available in command line mode.

ordinals

An exported function's ordinal is a two byte integer. The system loader will ultimately obtain the function's address from the ordinal table and the export address table. It does this by looking up the function's ordinal from its name and then using the ordinal as an index into the export address table. By default SLINK will assign ordinals to the exported functions. However, you may wish to guarantee that the function has the same ordinal in all builds of the DLL. In which case you may specify the ordinal with this command

e.g. the following example will assign ordinal 4 to the function **func** exported as **gloop**

```
export gloop=func @4
```


noname

This will export the function by ordinal only. This is used to hide a function within a DLL but still make it accessible to those who know its ordinal. You must specify the ordinal if you use the **noname** keyword.

e.g.

```
export func @4 NONAME
```

will export **func** by ordinal only, with an ordinal value of 4 whilst

```
export func NONAME
```

will produce an error.

exportall

Adds all exportable code entries to the export list. These are code symbols with storage class "external" that have been defined in a COFF object file, i.e. not a COFF archive or DLL. This excludes you from re-exporting an entry point in another DLL unless you specifically export it with the **EXPORT** statement.

exportx

This prevents symbols from being included in the export list generated by the **exportall** command. The **export** command will take precedence over this command if a symbol appears in both.

file (fi) filename

Performs the following actions in order.

- 1) Symbols specified in the export list are exported and the export (.edata) section generated. The archive, if required is also generated.
- 2) Scans the default libraries
 - a) FTN77.DLL or FTN95.DLL
 - b) SALFLIBC.LIB (unless an **entry** command has been used specifying other than **_SALFStartup** see above). SALFLIBC.LIB will be searched for in the following places:
 - i) Locally
 - ii) The directory specified in the environment variable **SCCLIB**.
 - iii) The directory above that specified in the environment variable **SCCINCLUDE**.
 - iv) The directory where the invoked copy of **SLINK** resides.

The scanning of each of the following DLLs is dependent upon there being unresolved references and upon the DLL in question being present. These are searched for in the following places:

- i) Locally

- ii) The “system directory”, i.e. the directory returned by the function **GetSystemDirectory** e.g. C:\WINNT\SYSTEM32.
- iii) The directories specified on the system path
- iv) The directory where the invoked copy of **SLINK** resides.
- c) **KERNEL32.DLL**
- d) **USER32.DLL**
- e) **GDI32.DLL**
- f) **COMDLG.DLL**
- 3) Generates the internal traceback map
- 4) Generates the map listing file if one has been requested.
- 5) Displays a list of unresolved external references.
- 6) Writes the executable. If an executable is to be written, a suffix of .EXE is appended to *filename* as a default if one has not been supplied. If a DLL is to be written, a suffix of .DLL is appended to *filename* as a default if one has not been supplied.
- 7) Exits **SLINK**.

Note: Missing externals will not cause a failure but will generate warnings. However, if an attempt is made to call one of the missing routines at run time a message will be printed out giving the name and the return address of the routine. The user's program is then aborted.

filealign value

Specifies the physical alignment of the sections within the file. *value* should be an integral power of 2.

default *value* = 0x200

heap reserve[,commit]

Specifies the program heap size in bytes. An initial heap of *commit* bytes will be allocated. If this is used up then a further *commit* bytes will be allocated up to the maximum size of *reserve*. The *reserve* and *commit* values are rounded to 4 byte boundaries.

Salford libraries provide their own heap and so a minimal heap need only be specified.

defaults:

for Salford programs:

reserve = 0x0

commit = 0x0

for other programs:

reserve = 0x100000 (1Mb)

commit = 0x1000 (4Kb)

imagealign (align) value

Specifies the virtual address alignment in bytes of sections within the executable. *value* should be an integral power of 2.

default *value* = 0x1000

imagebase (base) address

Specifies the preferred base address for the loaded image. This may be relocated by the loader.

The virtual address space begins at 0x00000000 but the user program starts much higher in memory. The **base** command specifies the virtual address at which the program is to start. This is called the preferred load address. If the system loader cannot load the program at that address it calculates what is called a **DELTA** which is the difference between the preferred load address and the actual load address. This **DELTA** is then applied to all the virtual addresses in the program, actually those specified in the program's fixup table. **SLINK** will set the base address to be 0x00400000 for executables and 0x01000000 for DLLs. You may wish to change the base address if there is already something else loaded at that address. The fixup process is much more likely to affect DLLs than executables. However, the fixup process is so fast that it is tiny in comparison with the load process overall and can safely be ignored. The base address is specified in decimal, but can also be specified in hex or octal using the 0x or 0 prefixes respectively.

The value set by the base command will be rounded down to be a multiple of 64K. The resulting value must be non zero.

defaults:

executables

address = 0x00400000

DLLs

address = 0x01000000

The following example sets the base address to be 0x00700000

```
BASE 0x700000
```

listunresolved (lure)

Prints a list of external references which are missing. This command may be used to check the progress of a **SLINK** session and may be used at any time. This command has no other effect.

load (lo) *filename*

Loads object file *filename*. *filename* may be either a COFF object file, a COFF archive library (i.e. .LIB) or a directly imported dynamic link library (.DLL). A .OBJ suffix will be appended to *filename* if one isn't already supplied.

SALFLIBC.LIB will automatically be loaded if has not already been loaded and if the entry point name has not been changed from SALFStartup.

map *filename*

Specifies that a symbol map file should be produced and written to *filename*. The action of this command will be deferred until all object files have been loaded. A suffix of .MAP is appended as a default if one has not been supplied.

notrace

Suppresses the generation of the internal map within the executable. Without this map a runtime traceback is impossible.

stack *reserve* [,*commit*]

Specifies the program stack size in bytes. An initial stack of *commit* bytes will be allocated. If this is used up then a further *commit* bytes will be allocated up to the maximum size of *reserve*. The *reserve* and *commit* values are rounded to 4 byte boundaries.

defaults,

for Salford programs:

reserve = 0x300000 (3Mb)

commit = 0x4000 (16Kb)

for other programs:

reserve = 0x100000 (1Mb)

commit = 0x1000 (4Kb)

virtualcommon (vc) [*base*, [*commit*]]

Specifies that the uninitialised data section, i.e. the .bss section is removed entirely from the executable and placed into virtual paged memory. The *base* address of this virtual memory may be specified but should be done with care. Similarly, a *commit* value can be specified to indicate how much memory should be committed from the system at each acquisition. Small values of *commit* mean that there is less memory wastage whilst larger values will improve (slightly) run time performance at the expense of memory usage.

base and *commit* must be aligned on a page boundary, i.e. if specifying the values in hex the least significant 3 digits must be zero.

defaults,

base = 0x20000000

commit = determined at run time initialisation

subsystem *subsys*

You should specify whether the program will require a Character User Interface (CUI) or a Graphical User Interface (GUI). The subsystem specified should be one of **console** for a CUI, **windows** for a GUI or **native** if no subsystem is required. Win32 will not allow output to **stdout** unless **console** has been selected as the subsystem.

By default, SLINK will set the subsystem to be **console**.

subsys should be one of the following:

native	no subsystem required
windows	a graphical user interface subsystem is required.
console	application requires only a character mode subsystem (but using a GUI is not precluded).

default: *subsys* = **console**

The following example sets a CUI subsystem requirement.

```
SUBSYSTEM CONSOLE
```

quit (q)

Immediately exits SLINK. No output files are produced.

Command Line mode

In this mode all of the object files and SLINK commands are placed on the command line.

```
SLINK [files] [options] [commandfile]
```

Object files, script files and options may be freely intermixed. There may be more than one command file.

1. Script files

Script files contain commands and/or object files.

Interactive style script files are prefixed by a “\$” or have no prefix. Command line script files are prefixed by an “@”.

2. Interactive style script files

These contain the same commands as may be used in interactive mode and are executed in the order that they appear in the file.

e.g.

```
SLINK myprog
```

If *myprog* (no filename extension) does not exist then *myprog* will be assumed a interactive style script file.

or

```
SLINK $myprog
```

This form is explicitly a interactive style script file.

3. Command line style script files and command line arguments

These may contain the following commands:

-addobj: { @ <i>listfile</i> <i>filename</i> }	
-align: #	
-archive: [<i>filename</i>]	alias for -implib
-base: <i>address</i>	
-debug [:full :partial :none]	
-decorate	
-dll [: <i>module</i> name]	alias for -library
-entry: <i>symbol</i>	
-export: <i>entryname</i> [=internalname]	
-exportall	
-exportx: <i>entryname</i>	
-file [: <i>filename</i>]	alias for -out
-filealign: #	
-heap: <i>reserve</i> [,commit]	
-help	alias for -?
-imagealign: #	alias for -align
-imagebase: <i>address</i>	alias for -base
-implib [: <i>filename</i>]	
-library [: <i>filename</i>]	
-map [: <i>filename</i>]	

-notrace
-out [:*filename*]
-stack:reserve[:,*commit*]
-subsystem: { **native** | **windows** | **console** }
-vc[:*baseaddress*[:,*commit*]] alias for **-virtualcommon**
-virtualcommon[:*baseaddress*[:,*commit*]] alias for **-vc**

Commands have the same requirements and meaning as the corresponding interactive command, may be in upper or lower case and may have either a “/” or a “-” prefix.

Commands are executed first and so may appear anywhere and in any order. Object files are scanned after all of the commands have been scanned.

Object file names are specified with the .OBJ extension and, unlike interactive mode no extension is automatically appended. However, object files are loaded in the order in which they appear.

The **-out**: (**-file**:) command is unnecessary since an executable will be automatically written after all commands have been processed and all object files have been scanned. For example:

```
SLINK myprog.obj -map: @others.inf -file:test @defltlib.inf
```

This will link the object file *myprog.obj* with any object files, libraries, or DLLs listed in *others.inf* or *defltlib.inf*. A map file *myprog.MAP* will be produced (name taken from the first loaded object) and executable *test.EXE* (automatic file name extension) written. For example:

```
SLINK tester.obj mylib.obj
```

tester.obj is scanned and then *mylib.obj* is scanned. An executable named *tester.EXE* will be written.

It is generally not advisable to mix interactive and command line style script files due to their different behaviour. If more than one interactive style script file is used remember that commands are executed in the order in which they appear.

Direct import of Dynamic Link Libraries

Care should be taken on importing DLLs. The mechanism is designed to replace the importation of “pure” import libraries. It is possible that the .LIB file contains loadable library code in addition to the imported symbol. In such cases, the loadable library code is missing from the DLL and so cannot be loaded. SALFLIBC.LIB is such a library, the DLL SALFLIBC.DLL cannot be loaded directly since it does not contain, for example, the symbol `_SALFStartup` which is necessary for initialisation and to provide the applications entry point.

Direct import of dynamic link libraries require that the exported names in the DLL follow the following rules

1) **__stdcall** symbols

The exported name is created by removing the leading underscore and does not contain the appended @ and subsequent characters

e.g. **_MessageBeep@4** will be exported as **MessageBeep**

2) Symbols beginning with a leading underscore

The leading underscore is removed.

3) Other symbols

All other symbols are assumed to be exported unchanged.

Archive and import library generation

Archives and import libraries may be generated without any objects being loaded with the **load** command.

For example the following **SLINK** script will generate a combined RLB and import library. Note that the **file** command is necessary to initiate the build. Ignore the comments in brackets

```
archive mylib           (archive file to be called mylib.LIB)
dll                    (module name set to mylib.DLL)
addobj func1.obj
addobj func2.obj
addobj func3.obj
addobj func4.obj
export functiona
export functionb
export functionc
export functiond
export functione
export functionf
export functiong
file
```

Entry Points

The executable file needs an entry point that is called by the system loader immediately after the program has been loaded. This entry point is not the main, **WinMain** or **LibMain** function as you may think, but library startup code. The Salford entry point for all executables, including DLLs is **_SALFStartup**. Object files

produced by other compilers will have a different entry point which you will have to set explicitly.

The entry point is specified with the **entry** command, omitting the leading underscore. SLINK will automatically set the entry point to be `_SALFStartup` unless you use the **entry** command *before* any objects files have been loaded. In command line mode the **entry** command can be placed anywhere in the command line or script file since commands are always processed first.

e.g. The following command will set the entry point to be `_SALFStartup`. This is redundant since SLINK will do this for you.

```
ENTRY SALFStartup
```

You cannot change the entry point after the first object file has been loaded or if an **entry** command has previously been used.

Using MK and MK32

Introduction

The Salford MK and MK32 utilities are similar to the UNIX MAKE program. Users who are familiar with MAKE should be able to use these utilities with little or no assistance. MK is a DOS/Win16 utility whilst MK32 is a Win32 utility.

A “make” utility is a project manager. Any given project is assumed to be based on a number of inter-related files. These might include a file for the main program, various files for the subroutines, various “include” files, object files, libraries, and maybe a final executable file. These are assumed to be inter-dependent, in that a change in one file will have repercussions on other files. For example a change in an “include” file will affect any source file which uses that include file, this in turn will affect the resulting object files and so on.

The purpose of a “make” utility is to read a file which describes all of the inter-dependencies in a given project and update only those files that need to be updated. The updating is based on the given dependency relationships and also on the current date/time stamp for the files. Thus if file A is given to be dependent on file B and file A predates file B, then file A is updated.

Note that a “make” utility uses the date/time stamp that the operating system places on a file when it is saved. If the computer date/time is not functioning correctly then the utility is unlikely to have the desired effect.

Tutorial

In order to illustrate how this works, we shall consider the following simple situation.

Suppose we have a project based initially on two files. The first file, called *prog.for*, contains only a main program; the second *sub.for* contains all of the user-defined subroutines that are called from the main program. In all other respects, these files are assumed to be independent of each other and independent of any other user files.

The simplest way of calling the Salford make utility is to type just MK (or MK32) at the command prompt. If you do this then the utility processes a file called *makefile* which the user places in the current directory. *makefile* contains dependency relationships and dependency rules (either explicit or implicit) for the current project.

Example 1

This first example illustrates the use of explicit dependency relations.

In the project described above, *makefile* could contain:

```
prog.exe: prog.obj sub.obj
        SLINK prog.lnk

prog.obj: prog.for
        FTN77 prog /check

sub.obj:  sub.for
        FTN77 sub /check
```

This means that *prog.exe* depends on both *prog.obj* and *sub.obj* and that *prog.exe* is created by calling **SLINK** using *prog.lnk* as the linker script. For DOS/Win16 you would use **LINK77** instead.

In turn *prog.obj* depends on *prog.for* and *prog.obj* is created by calling **FTN77** using *prog.for* with the **/CHECK** option. A similar dependency relationship and rule is used for *sub.obj*.

If *sub.for* only were changed (for example) then calling the utility would result in *sub.for* being recompiled (but not *prog.for*). Then because *prog.exe* depends on *sub.obj*, the linking process is also carried out.

Example 2

A second approach is to use an implicit dependency relationship as illustrated here:

```
.SUFFIXES: .for .obj
```

```
.for.obj:
    FTN77 $< /check

prog.exe: prog.obj sub.obj
    SLINK prog.lnk
```

The explicit dependency relation for *prog.exe* has not changed. The first line gives a list of extensions (separated by at least one space) for which implicit relations will be supplied. In this case one relation is given showing how “.obj” files are derived from “.for” files.

The next line is an example of an implicit relation. In this case the relationship states that (in the absence of an explicit relation) a “.obj” file is dependent on a corresponding “.for” file and that the object file is formed by calling FTN77 with the /CHECK option. “\$<” represents the source filename (the dependency filename with its extension). In other words, we have now used one implicit relation in place of two explicit relations in Example 1.

Example 3

Our next example of a *makefile* illustrates a use for the TOUCH utility and takes the form:

```
.SUFFIXES: .for .obj

.for.obj:
    FTN77 $< /check

prog.exe: prog.lnk
    SLINK prog.lnk

prog.lnk: prog.obj sub.obj
    touch prog.lnk
```

The TOUCH utility simply updates the date/time stamp of the given file. So here we are saying that *prog.lnk* depends on *prog.obj* and *sub.obj* but the content of *prog.lnk* does not need to be changed. The order of the two explicit relations is significant; *prog.exe* is the primary target and must come first.

Example 4

Now we take example 3 one stage further:

```
.SUFFIXES: .for .obj .lnk .exe

.for.obj:
    FTN77 $< /check
```

```
.lnk.exe:
    SLINK $<

prog.exe:

prog.lnk: prog.obj sub.obj
    TOUCH prog.lnk
```

This includes an implicit relation which connects the linker script to the executable. The result is neither shorter nor simpler than example 3 and so has little merit unless you also use a *default.mk* file (see below).

Example 5

We now return to the form given in example 2 and provide a modification which illustrates the use of macros and comments:

```
# Example 5

.SUFFIXES: .for .obj

OBJFILES=prog.obj \
    sub.obj
T=prog

.for.obj:
    FTN77 $< /check

$T.exe: $(OBJFILES)
    SLINK $T.lnk
```

Characters after a “#” symbol on a given line are ignored so the first line is a comment. **OBJFILES** and **T** are macros. They represent constant character strings which replace expressions of the form $$(...)$ within dependency relations. If the macro name consists of only one character then the parenthesis is not required. The backslash (\) character is used for continuation (suppressing the following carriage return/linefeed). The following macros are implicitly defined:

\$@	evaluates to the file name of the current target
\$*	evaluates to the file name of the current target without its extension
\$<	evaluates to the source filename in an implicit rule

In a macro assignment, spaces can be used on either side of the equals sign. Macro names are case sensitive although it is common to use only upper case letters. Also, it is possible to append a string to an existing name as follows:

```
OBJFILES=prog.obj
OBJFILES+= sub.obj
```

but note that the space before *sub.obj* is essential in this context.

Example 6

Our next example is similar to example 4 but illustrates the use of a file called *default.mk*. Create a file in the project directory called *default.mk* containing the implicit relations:

```
.SUFFIXES: .for .obj .lnk .exe

.for.obj:
    FTN77 $< /check

.lnk.exe:
    SLINK $<
```

makefile now contains:

```
prog.exe:

prog.lnk: prog.obj sub.obj
    TOUCH prog.lnk
```

MK/MK32 automatically calls *default.mk* and uses it as a header.

You will find a file called *default.mk* in the compiler directory. This file can be copied to your project directory and customised to suit your particular project.

If you wanted to include something other than (or as well as) *default.mk* in your *makefile* then insert a line of the form:

```
include filename
```

There must be no spaces at the beginning of this line.

Example 7

Taking this one stage further we now include macros in *default.mk*:

```
.SUFFIXES: .for .obj .exe

OPTIONS=

OBJFILES=
```

```
.for .obj:
    FTN77 $< $(OPTIONS)

.obj.exe:
    SLINK $(OBJFILES) -FILE:$@
```

This uses the command line form of **SLINK** which is not available with **LINK77**. *makefile* now contains:

```
OPTIONS=/check

OBJFILES=prog.obj sub.obj

prog.exe: $(OBJFILES)
```

Example 8

Our final example uses the same *default.mk* as in example 7 but moves **OPTIONS** and a new macro called **TARGET** to the command line. *makefile* now contains:

```
OBJFILES=$(TARGET).obj sub.obj

$(TARGET).exe: $(OBJFILES)
```

and the command line takes the form:

```
MK32 TARGET=prog OPTIONS=/check
```

Macros that are defined on the command line replace any definitions that appear in the makefiles. Alternatively you could define **TARGET** and/or **OPTIONS** as DOS environment variables. Other items that can be added to the command line are given below.

Reference

Command line options

- | | |
|--------------------|---|
| -f <i>filename</i> | Use <i>filename</i> instead of the default file called <i>makefile</i> . A minus sign in place of <i>filename</i> denotes the standard input. |
| -d | Display the reasons why MK/MK32 chooses to rebuild a target. All dependencies which are newer are displayed |

-dd	Display the dependency checks in more detail. Dependencies which are older are displayed, as well as newer.
-D	Display the text of the makefiles as they are read in.
-DD	Display the text of the makefiles and <i>default.mk</i> .
-e	Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macros definitions.
-i	Ignore error codes returned by commands. This is equivalent to the special target <i>.IGNORE:</i> .
-n	No execution mode. Print commands, but do not execute them. Even lines beginning with an @ (see Rules below) are printed. However, if a command line is an invocation of MK/MK32, that line is always executed.
-r	Do not read in the default file <i>default.mk</i> .
-s	Silent mode. Do not print command lines before executing them. This is equivalent to the special target <i>.SILENT:</i> .
-t	Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.
macro=value	Macro definition. This definition remains fixed for the MK/MK32 invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate MK/MK32's but acts as an environment variable for these. That is, depending on the -e setting, it may be overridden by a makefile definition.

Makefiles

The first makefile read is *default.mk*, which can be located anywhere along the PATH. It typically contains pre-defined macros and implicit rules.

The default name of the makefile is *makefile* in the current directory. An alternative makefile can be specified using one or more -f options on the command line. Multiple -f's act as the concatenation of all the makefiles in a left-to-right order.

The makefile(s) may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by using backslash (\) at the end of a line.

Anything after a “#” is considered to be a comment. Completely blank lines are ignored.

An include line is used to insert the text of another makefile. It consists of the word “include” left justified, followed by spaces, and followed by the name of the file that is to be included at this line. Include files may be nested.

Macros

Macros have the form `WORD=text`. `WORD` is case sensitive although commonly upper case. Later lines which contain `$(WORD)` or `${WORD}` will have this replaced by ‘text’. If the macro name is a single character, the parentheses are optional. The expansion is done recursively, so the body of a macro may contain other macro invocations. Spaces around the equal sign are not relevant when defining a macro. Macros may be extended to by using the “+=” notation.

Special macros

MAKEFLAGS This macro is set to the options (not macros) provided on the command line for MK/MK32. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested calls to MK/MK32, but it is also available to these invocations as an environment variable.

SUFFIXES This contains the default list of suffixes supplied to the special target `.SUFFIXES:`. It is not sufficient to simply change this macro in order to change the `.SUFFIXES:` list. That target must be specified in your makefile.

- \$*** The base name of the current target (used in implicit rules).
- \$<** The name of the current dependency file (used in implicit rules).
- \$@** The name of the current target.

Targets

The form of an explicit dependency rule is:

```
target [target] [. . .]: [source] [. . .]  
    [rule]  
    [. . .]
```

Here we have one or more target files, each separated by a space, and followed by a colon (there must be no spaces before the first target). Then we have zero or more dependent files followed by zero or more rules (see below), each on its own line preceded by at least one space (again the space is essential). See example 2. The targets can be macros that expand to targets.

The colon that appears in a dependency rule does not interfere with a colon that appears in the path of a file (after the drive letter).

If a target is named in more than one target line, the dependencies and rules are added to form the target's complete dependency list and rule list.

The dependants are ones from which a target is constructed. They in turn may be targets of other dependants. In general, for a particular target file, each of its dependent files is 'made', to make sure that each is up to date with respect to its dependants.

The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependants have changed, so the target must be constructed. This checking is done recursively, so that all dependants of dependants etc. . . are up to date.

To reconstruct a target, MK/MK32 expands macros and either executes the rules directly, or passes each to a shell or COMMAND.COM for execution.

For target lines, macros are expanded on input. On other lines macros are expanded at the point of implementation.

Special targets

.DEFAULT:

The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. MK/MK32 ignores all dependencies for this target.

.DONE:

This target and its dependencies are processed after all other targets are built.

.IGNORE:

Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying -i on the command line.

.INIT:

This target and its dependencies are processed before any other targets are processed.

.SILENT:

Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying -s on the command line.

.SUFFIXES:

The suffixes list for selecting implicit rules. Specifying this target with dependants adds these to the end of the suffixes list. Specifying it with no dependants clears the list. In order to add dependants to the head of the list, use the form:

```
.SUFFIXES: .abc $(SUFFIXES)
```

Rules

A line in a makefile that starts with a TAB or SPACE is a rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. Rules may have any combination of the following characters to the left of the command:

@ will not echo the command line.

- MK/MK32 will ignore the exit code of the command, i.e. the ERRORLEVEL of MSDOS. Without this, MK/MK32 terminates when a non-zero exit code is returned.
- + MK/MK32 will use COMMAND.COM to execute the command. If the '+' is not attached to a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway.

Implicit rules

The form of an implicit rule is:

```
.source_extension.target_extension:
    [rule]
    [. . .]
```

Here we have a dot (no spaces before it) followed by the extension for the source file (one, two or three characters) then a dot followed by the extension for the target file and then a colon and then a new line and at least one space. Optional rules then follow on separate lines just as in an explicit dependency. See example 3.

Implicit rules are linked to the .SUFFIXES: special target. Each entry in .SUFFIXES defines an extension to a filename which may be used to build another file. The implicit rules then define how to build one file from another. These files are related, in that they must share a common base name, but have different extensions.

If a file that is being made does not have an explicit target line, a search is made for an implicit rule. Each entry in the .SUFFIXES: list is combined with the extension of the target, to get the name of an implicit target. If this target exists, it gives the rules used to transform a file with the dependent extension to the target file. Any dependants of the implicit target are ignored.

Files

makefile	Current version(s) of make description file.
default.mk	Default file for user-defined targets, macros, and implicit rules.

Diagnostics

MK/MK32 returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

Badly formed macro

A macro definition has been encountered which has incorrect syntax. Most likely, the name is missing.

Cannot open file

The makefile indicated in an include directive was not found or was not accessible.

Don't know how to make target

There is no makefile entry for target, none of MK/MK32's implicit rules apply, and there is no .DEFAULT: rule.

Improper macro

An error has occurred during macro expansion. The most likely error is a missing closing bracket.

Rules must be after target

A makefile syntax error, where a line beginning with a SPACE or TAB has been encountered before a target line.

Too many options

MK/MK32 has run out of allocated space while processing command line options or a target list.

Using Plato

Introduction

Plato is a Win32 editor which supports all of Salford's Win32 compilers. It is a multiple document interface with many features including syntax colouring, unlimited undo and keyword help.

This chapter describes how to use Plato and shows how it is possible to compile, link and execute Salford programs from within an integrated development environment.

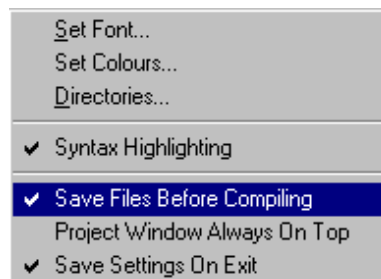
Getting started

Run Plato by clicking on the Plato shortcut icon in the "Salford Software" program group.



Before proceeding further check that Plato has the correct location of you compilers and help files. To do this select **Directories** from the **Options Menu**.

The Options Menu



This will display a window showing the directory location of all Salford Compilers and Help files. If you did not install your compiler(s) to the default directories you may need to change the paths using the **Browse** button. Click the **Apply** button to update your changes. Make sure **Save Settings On Exit** is selected as above to ensure any configurations you have made are saved.

The toolbar at the top of the Plato screen controls most of Plato' commonly used functions.

The Toolbar



New File

This button has the same effect as the **New** command on the **File** menu. It opens a new blank Edit Window ready to begin typing a new source file.



Open File

This button has the same effect as the **Open** command on the **File** menu. It presents a standard 'Open File' dialog and prompts the user to select an existing source file. The filename the user provides will then be opened in a new Edit Window.

Filenames ending in *.c or *.cpp are assumed to be C++ files; *.for, Fortran 77 files and *.f90, Fortran 90 files. You can change the compiler associated with an open file - see **Changing File Options**.



Save File

This button has the same effect as the **Save** command on the **File** menu. It saves the active source file with the current filename. If no filename has been assigned the user is prompted to enter one.



Print File

This button has the same effect as the **Print** command on the **File** menu. The file is sent directly to the current printer, which can be configured from the **File** menu.



Cut

This button has the same effect as the **Cut** command on the **Edit** menu. It removes the selection from the active source file and places it on the clipboard.

**Copy**

This button has the same effect as the **Copy** command on the **Edit** menu. It copies the selection onto the clipboard.

**Paste**

This button has the same effect as the **Paste** command on the **Edit** menu. It inserts the contents of the clipboard at the insertion point replacing any selection.

**Undo**

This button allows you to undo previous editing instructions.

**Find**

This button has the same effect as the **Find** command on the **Edit** menu. It searches for specified text in the active source file.

**Compile File**

This button has the same effect as the **Compile File** command on the **Project** menu. It compiles the active source file.

**Build File**

This button has the same effect as the **Build File** command on the **Project** menu. It compiles and links the active source file.

**Compile Project**

This button has the same effect as the **Compile Project** command on the **Project** menu. It compiles all modified source files in current project.

**Build Project**

This button has the same effect as the **Build Project** command on the **Project** menu. It compiles all modified source files and links the current project.

**Rebuild Project**

This button has the same effect as the **Rebuild All** command on the **Project** menu. It compiles and links all files in the current project.

**Execute**

This button has the same effect as the **Execute** command on the **Project** menu. It runs the last file or project built.

**Debugger**

This button launches the Salford debugger SDBG and is available when you have built an executable, which should have been compiled with debugging options.

**Show Error Window**


This button displays the error window which displays errors, warnings or comments generated from the last compile.


The toolbar also has a pull down listbox containing the files that are at present open or part of an open project. You can switch between windows by selecting a filename from this listbox.

Editing Source Files


You can edit compile and run individual source files using buttons from the toolbar:

Creating a New File

Select the **New** command from the **File** menu or click  which opens a new edit window in which you can type in your program.


The edit window will be labelled *Untitled* (That is your source file has not been assigned a file name). When you now click the save button  a **Save As** dialog box will be presented. Use the **Save As** dialog box to navigate your disk and find an appropriate folder to in which save the source file. Type a file name for your source file and then click the **Save** button. Your source file will be saved to disk. Make sure to use the appropriate extension for your file otherwise Plato will not know which compiler to use.

Open an Existing File

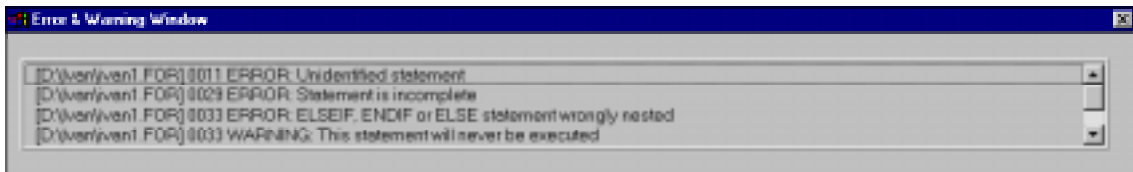
Select the **Open** command from the **File** menu or click  This presents a standard dialog similar to the File Open dialog of many windows applications. Use the dialog box to navigate your folders and select the file you want to edit then click the **Open** button. The existing source file will be opened into a new MDI Edit Window.



When a file is opened, the name is recorded in the Most Recently Used (MRU) list on the file menu, this list is saved and restored the next time Plato is started. The file name is also placed into a drop down list box on the toolbar.

Compiling a Single Source File.

- 1) Select the **Compile File** command from the **Project** menu or click . A dialog box will appear while the source file is compiled.
- 2) When the file is compiled the Compilation Status window will appear showing the number of errors, warnings and comments. If the compile has been successful the Compilation Status Icon will turn green, if not it will turn red.
- 3) Click the **Details** button to open the Message window and view any errors, warnings and comments. You can quickly moved to the line where an error occurred by double clicking on the appropriate line in the Message window.

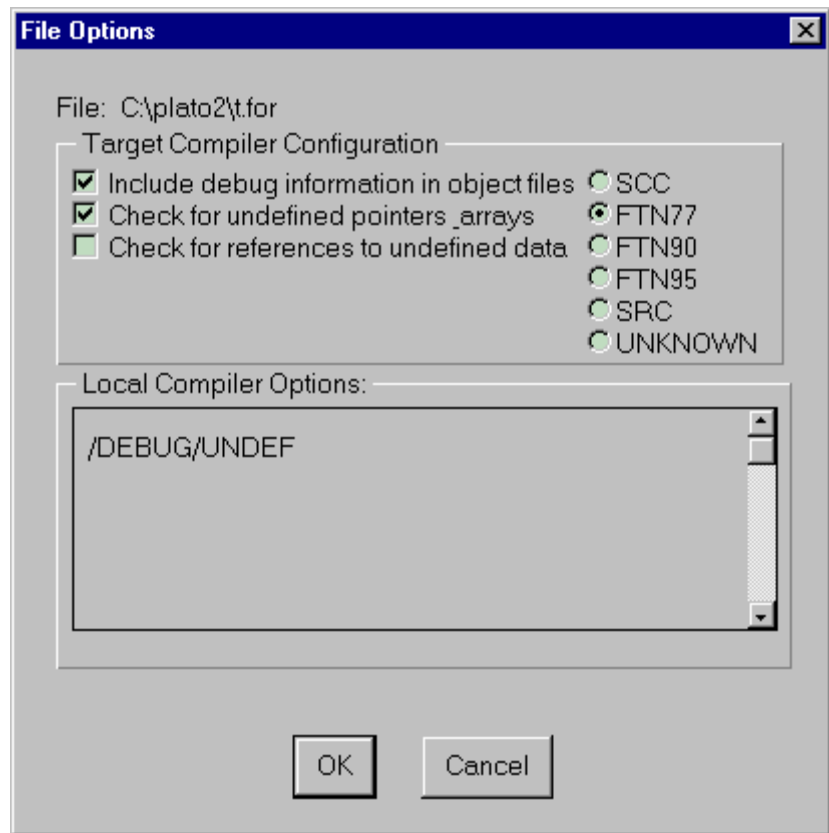
The Message Window



- 4) Once you are happy with your compilation, choose build  which will link your program.
- 5) If the linking is successful choose **Run** (or click **OK** and then the  toolbar button). A window appears showing you the file to be executed and two radio buttons. If you select **Console**, Plato will open up a console before running the file.

Changing File Options

You add compiler options or change the compiler associated with the currently opened file by selecting **File Options** from the **File** menu. The check boxes provide quick access to common options whilst the edit box below it allows any option to be entered.



Working with Projects

One of Plato's major features is the ability to organise the source files that make up your program into projects. Along with other benefits this enables you to compile and link all your sources in one go.

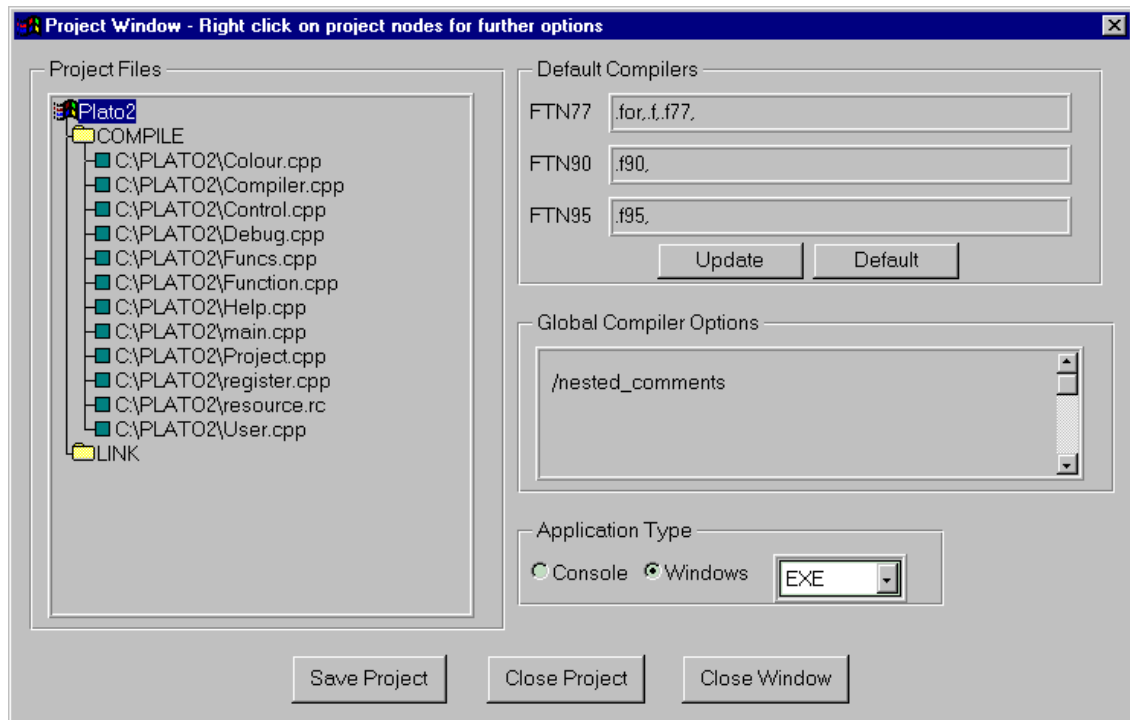
Creating a New Project

To create a new project follow these steps:

- 1) Click **New** from the **Project menu** to open an empty project window.
- 2) Give the project a suitable title by clicking the left and then *right* mouse button on the project icon at the top of the tree and selecting **Project Name**.
- 3) Build a list of source files. Left then *Right* click on the **COMPILE** folder in the Application tree and select **Add Item**.

- 4) Use the dialog box to navigate your folders and select the source file(s) that are part of this application then click the **Open** button. The source file(s) you have selected will be displayed in the Application tree.

The Project Window



- 6) To specify compiler options for a particular file click the right mouse button on the filename and choose **File options** from the popup menu. To supply compiler options that will affect all files use the Global Compiler Options edit box.
- 7) Save the project with the **Save Project** button.
- 8) Double click a file in the Application tree to open it for editing and click the **Close Window** button. You can return to the project window from the Project Menu.

Compiling and Building a Project

Compiling a project is similar to compiling single files, you can use the toolbar to Compile, Build and Rebuild your projects.

The Project Menu

<u>C</u> ompile File	F9
<u>B</u> uild File	Alt+F9
<u>C</u> ompile Project	F8
<u>B</u> uild Project	Alt+F8
<u>R</u> ebuild Project	Shift+F8
<u>E</u> xecute	Ctrl+F5
<u>N</u> ew	
<u>O</u> pen...	
<u>S</u> ave	
Save <u>A</u> s...	
<u>C</u> lose	
Project <u>W</u> indow	
Error Messages <u>W</u> indow	

When a project is built, all the files in the application tree are processed and any files that do not have an up to date object file are compiled.

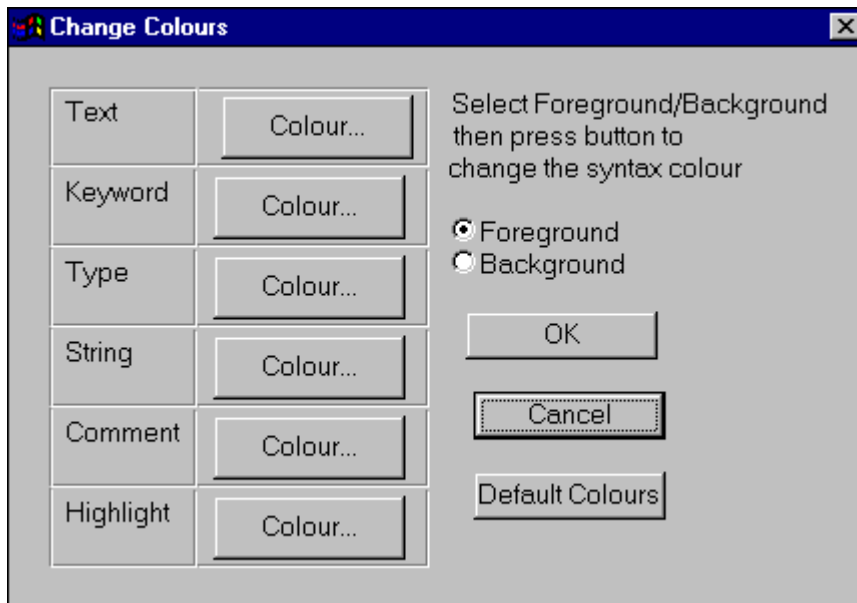
Projects - Advanced Features

Since there are many different Fortran file extensions and three different Salford Fortran compilers, Plato allows you to choose which of these compilers is associated with user specified file extensions for your project. From the project window edit the strings in the Default Compilers section and press the **Update** button.

The Project Window allows you to create DLLs (Dynamic Link Libraries) and RLBs (Relocatable Binary Libraries). Select from the pulldown list box in the Application Type section.

Customising Plato

You can change the font used to edit files by selecting **Set Font** from the Options Menu. In addition you can change the colours associated with different program elements by selecting **Change Colours** which is also in the Options Menu.



Keywords are those words defined by the compiler you are using, e.g. `PRINT` in Fortran and `printf` in C. Types include `INTEGER` and `REAL` in Fortran and `static` and `int` in C.

Accelerator Keys

Standard Windows

Key	Action
Ctrl+N	Creates a new edit window
Ctrl+O	Opens a file
Ctrl+S	Saves the current file
Ctrl+P	Prints a file
Ctrl+Z	Undo
Ctrl+X	Cut
Shift+Del	Cut
Ctrl+C	Copy

Ctrl+Ins	Copy
Ctrl+V	Paste
Shift+Ins	Paste
Ctrl+A	Select all
Ctrl+F	Find
Ctrl+H	Find and replace
Ctrl+G	Go to line
F1	Help topics
Shift+F1	Keyword help

Compiling

Key	Action
F2	Save
F3	Save and close file
F4	Close file
F5	Find string
F6	Replace string
F9	Compile file
Alt+F9	Build file
F8	Compile project
Alt+F8	Build project
Shift+F8	Rebuild project
Ctrl+F5	Execute
Alt-F2	Save project
F10	Project properties
Ctrl+F1	Keyword help

Block Marking

Key	Action
Alt+B	Mark block
Alt+- (minus)	Cut block
Alt-L	Mark line
Alt-Z	Paste block

DBOS (DOS)

Introduction

When a 32-bit Intel chip is running MS-DOS it operates in “real mode”. In this mode the chip emulates an 8086 chip with the addition of a few extra instructions and, of course, a much improved performance.

Real mode offers a 1 Megabyte address space composed of 64K byte segments and no protection of the operating system. In real mode, RAM at addresses above 1 Megabyte is unaddressable. In order to gain access to these locations and use FTN77, it is necessary to run DBOS. DBOS is the DOS extender provided with FTN77. Its purpose is to provide services to applications compiled using FTN77 and her sister compilers. The majority of the DBOS system stays resident above DOS. A small portion (approximately 45 Kilobytes) stays resident in real mode memory and provides services via an interrupt (78 hex).

The main service that is provided is to switch the program from real mode to protected mode. This is the first operation performed by all programs compiled with FTN77. When the program terminates, DBOS returns the processor to real mode and returns control to MS-DOS.

Switching to protected mode allows access to the extended memory above the 1 Megabyte limit. It also opens up the possibility of clashes between DBOS and other applications such as other memory managers or disk caches.

DBOS must not be used at the same time as other software which exploits extended memory unless it is compatible with the Virtual Control Program Interface (VCPI) eXtended Memory Specification (XMS) or an interrupt 15 top down allocator.

This restriction is necessary because DBOS uses all the available extended memory for user page space. This can cause problems if there are other programs which also use extended memory on your machine. In particular, the problems caused by DBOS overwriting a disk-cache's data area are usually catastrophic.

However, DBOS may be used with software that uses extended memory if that software supports the VCPI (for example QEMM386 from Quarterdeck), VDISK 4.0 or XMS (also known as the HIMEM.SYS scheme). DBOS recognises programs which use these protocols without overwriting their associated memory areas.

The “VDISK” method uses the fact that a subfunction of interrupt 15 (hex) returns the number of kilobytes of extended memory available from address 100000 (hex). By default DBOS hooks this interrupt so it returns a smaller value.

DBOS and expanded memory managers

Expanded memory managers (EMMs) emulate expanded memory boards in order to provide expanded memory specification (EMS) memory. Some also provide the VCPI so that programs using extended memory and protected mode can allocate memory and switch in and out of protected mode. DBOS can use this latter kind of EMM.

EMMs come in two basic types, *common pool* and *separate pool* providers.

Common pool EMMs provide XMS memory, VCPI memory and EMS memory from a single memory pool. No configuration of this pool is required by the user. Two well known common pool providers are QEMM386 from Quarterdeck and 386MAX/BlueMAX from Qualitas. DRDOS 6.0's EMM386 is also a common pool provider. DBOS will run with common pool providers and will use the VCPI and VCPI memory.

Separate pool providers maintain separate pools for EMS/VCPI memory and XMS memory. This is usually because the XMS provider and the EMS/VCPI provider are separate pieces of software e.g. HIMEM.SYS (XMS provider) and EMM386 (EMS/VCPI provider). Separate pool providers require you to explicitly transfer memory from the XMS pool to the EMS/VCPI pool. MS-DOS 5.0's HIMEM.SYS and EMM386.EXE and Compaq's HIMEM.EXE and CEMM.EXE are separate pool providers.

VCPI memory is demand paged. That is, DBOS uses the VCPI to allocate memory from the pool as programs demand it. Memory is released to the pool as programs release memory and terminate.

XMS memory is not demand paged, instead, DBOS will use all the available XMS memory (less that which may have been reserved using the /EXTMEM command line option). For example using,

```
DBOS /EXTMEM 400000
```

will leave 4Mb of XMS memory for use with non-DBOS applications, and allocate the remaining XMS memory for use with DBOS applications.

DBOS versions prior to 2.70 cannot use XMS memory if an EMM is installed. DBOS versions after 2.51 may use XMS memory provided that an EMM is not installed. Versions of DBOS after 2.70 can have access to XMS even after an EMM is installed.

Removing DBOS (using the KILL_DBOS utility program) will release all the memory that DBOS has reserved. You should ensure that other TSR (terminate and stay resident) programs are loaded before DBOS. In particular, the user should note that some MS-DOS commands load a memory resident portion (e.g. PRINT, MODE and APPEND) and the first use of such commands should occur before DBOS is loaded.

Note:

If an EMM has been installed you will initially be in virtual mode at the DOS command line prompt, and DBOS will use the VCPI to switch in and out of protected mode, regardless of whether VCPI or XMS memory is being used.

DBOS versions prior to 2.70

Common pool providers

Common pool providers need no special configuration for use with DBOS beyond the usual include/exclude list on the EMM command line.

Separate pool providers

Separate pool providers require the transfer of memory to the EMS/VCPI pool from the XMS pool. For example, a command line similar to the one below, for use with MS-DOS 5.0 HIMEM.SYS and EMM386 will provide 4096Kb of memory to the EMS/VCPI interface for use as VCPI memory with DBOS.

```
DEVICE=C:\DOS\HIMEM.SYS  
DEVICE=C:\DOS\EMM386.EXE 4096 RAM
```

DBOS versions 2.70 and above

Common pool providers

DBOS detects the presence of a common pool provider and by default will use the VCPI to allocate memory. This default behaviour can be changed by using the /USE_XMS DBOS command line option, in which case DBOS will use XMS memory for programs but still use the VCPI for virtual/protected mode switches. However, there is no advantage to using XMS memory in this situation. In fact, some common pool EMMs provide less XMS memory than VCPI memory.

Separate pool providers

If you have partitioned your memory between XMS and EMS/VCPI, by default DBOS will allocate memory from the largest pool. This behaviour can be changed by using the /USE_VCPI or the /USE_XMS command line option to force the use of a particular memory pool. However, unless you need to use VCPI or EMS memory for some other application, you need not allocate any memory to your EMM at all. For example, with MS-DOS 5.0,

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE NOEMS
```

is a suitable combination for DBOS and will provide the maximum amount of XMS memory to the system whilst still giving access to the upper memory block (UMB, that which is between 640Kb and 1 Mb) for loading TSR's and device drivers high. It also has the advantage of not requiring a page frame thus making a further 64Kb of UMB space available.

Here are some further examples:

Assuming an 8Mb machine (i.e. 7Mb of extended memory) and MS-DOS 5.0, then with

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE NOEMS
```

all of the extended memory available from XMS will be used by DBOS. Similarly with

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE 2048 RAM
```

2Mb of extended memory will be allocated to EMS/VCPI leaving approximately 5Mb available for XMS. In this case, DBOS will use the XMS memory pool since it is larger. The UMB area will be reduced by 64Kb by the need for a page frame. As a further example if you set:

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE 4096 RAM
```

then 4Mb of extended memory will be allocated to EMS/VCPI leaving approximately 3Mb available for XMS. In this case, DBOS will use the VCPI memory pool since it is larger. The UMB area will be reduced by 64Kb by the need for a page frame.

Network cards and expanded memory managers

In general, EMMs need to be told which regions of the UMB space are not available to them. These areas must be explicitly excluded on the EMM command line. The most common area that has to be excluded is the hardwired buffer on a network card. This is usually 8Kb long and resides in the D000-DFFF region. If the system exhibits instability when a network driver is loaded then it is likely that the network buffer has not been excluded from the EMM. Typically, with MS-DOS 5.0 the following may cure the problem:

```
DEVICE=C:\DOS\HIMEM.SYS  
DEVICE=C:\DOS\EMM386.EXE NOEMS X=D000-D1FF
```

If it does not, try excluding the whole of the D0 segment as below,

```
DEVICE=C:\DOS\HIMEM.SYS  
DEVICE=C:\DOS\EMM386.EXE NOEMS X=D000-DFFF
```

Use of a network card may require that other areas need to be excluded from the EMM.

DBOS command line arguments

In order to provide the services described above, DBOS must already have been loaded into the system. This is done by typing DBOS, possibly followed by command line arguments. DBOS will then load into memory, determine the operating environment and return to the MS-DOS prompt leaving itself in memory. DBOS is a Terminate and Stay Resident (TSR) program. DBOS can be removed at a later stage by typing the command KILL_DBOS.

The options available on the DBOS command line are:

/EXTMEM <h>

Preserves <h> bytes of extended memory for other programs. This is achieved by lowering the amount interrupt 15 returns to a value which preserves the amount of memory specified. Thus the memory preserved is at the bottom of physical extended memory, typically from 1 megabyte onwards. <h> must be specified in hexadecimal. For example:

```
DBOS /EXTMEM 100000
```

would leave 1 megabyte of extended memory free.

/SEARCHMEM

Instructs DBOS not to use the VDISK mechanism for determining the amount of

available extended memory, but to search for available memory. This option is especially useful with early Compaq machines which remapped some of the memory between 640KB and 1MB to just below 16MB. The VDISK mechanism would not allow this memory to be used, /SEARCHMEM will, however, find it.

/PAGE <h>

This option is designed for use in multi-tasking environments such as that provided by DESQview. The option tells DBOS to only use <h> bytes of memory before paging to disk. In the first instance paging must be enabled by configuring DBOS using the CONFIGDB utility. But even when this has been done, by default paging is disabled when running under DESQview. This option has the additional effect re-enabling paging when inside DESQview. <h> must be given in hexadecimal.

/NO_SHIFT_INTERRUPTS or /NSI

From version 2.69 onwards a former option /SHIFT_INTERRUPTS became the default. /NO_SHIFT_INTERRUPTS (abbreviated to /NSI) can be selected in the unlikely event that you wish to bypass this action.

/SHIFT_INTERRUPTS remaps the IRQs used by DBOS to avoid collision with other hardware. This prevents a clash between a piece of hardware (typically a BUS mouse or networking card) and an interrupt that DBOS uses.

/NOWEITEK

For DBOS versions before 2.76 this option prevents DBOS from checking the presence of a Weitek 1167, 3167 or 4167 numeric coprocessor. Some early 80386 machines have hardware problems which can cause strange behaviour when a Weitek coprocessor is accessed. If you have such a machine it is possible a Weitek coprocessor will not work in your machine.

/WEITEK

For DBOS versions from 2.76 onwards /WEITEK must be explicitly specified in order to enable DBOS to check for the presence of a Weitek coprocessor. This option is required in order to run an executable that was compiled using the FTN77 /WEITEK command line option.

/PRIMELINK

Allows DBOS to reside with Primelink, a terminal emulation package from PRIME.

/DISK_CACHE

Enables the DBOS disk-cache. This has the advantage of being DBOS compatible and making efficient use of extended memory. The disk-cache will use free memory pages. This is very flexible, allowing DBOS to take pages from the cache and give them to an FTN77 program. In this way the disk-cache size varies depending on the task being performed. A program in the compiler directory called CACHESTATS can be run to give statistics about the disk-cache.

The disk-cache is not available if an EMM is installed. However, all programs, including non-DBOS applications, can obtain the benefit of the DBOS disc-cache when it is installed.

/USE_VCPI or /UV

/USE_XMS or /UX

These options are for use with extended memory managers (EMMs) that support XMS and VCPI. Some EMMs have a common pool for XMS and VCPI memory whilst others have separate pools for each type (see page 302).

For common pool EMMs, the default DBOS behaviour is to use VCPI memory and the **/USE_XMS** option should only be selected if you wish to force the use of XMS memory.

For separate pool EMMs, the default DBOS behaviour is to use the larger of the two pools. Selecting **/USE_VCPI** or **/USE_XMS** will over-ride the default behaviour.

/CFG <config_file_path>

Specifies the path to an alternative DBOS configuration file. This allows the possibility of several configurations for DBOS. One possible use of it is to specify a configuration file which would cause DBOS to use a different paging file. This is particularly useful under Windows, where several DBOS invocations are simultaneously possible, only one of which may use the default paging file.

In addition to the options described above, DBOS can be invoked with an explicit memory range. The memory limit or limits must be given before any other options.

This can take one of two forms:

DBOS <upper_limit>

Ensures that DBOS will not use any memory above and including the physical address **<upper_limit>**. This value must be in hexadecimal. For example:

```
DBOS 100000
```

will prevent DBOS from using any extended memory (100000 hex is 1 megabyte).

DBOS <lower_limit> <upper_limit>

This form of memory specification can be used to circumvent a problem which occurs with large amounts of memory. If for example, a machine has 32 megabytes of memory, there are some BIOSs which will only recognise the first 16 megabytes of memory. This can cause two sorts of problem:

Firstly, interrupt 15 will only report that there is 16 megabytes in the machine.

Secondly, there will be a problem if another program has already taken some memory away (the BIOS will quite often do this), leaving a small amount of space that should not be touched just below 16Mb.

In this context, **DBOS** has two differing behaviours, depending upon the relationship of the physical address `<lower_limit>` to the amount of memory reported by the interrupt 15 mechanism.

- a) If `<lower_limit>` is less than the amount of memory reported by interrupt 15 then **DBOS** will take `<lower_limit>` as the physical lower limit of the memory it is to use. That is, **DBOS** will only use memory at physical addresses above this limit.
- b) If `<lower_limit>` is greater than the amount of memory reported by interrupt 15 then **DBOS** assumes that the machine is one with more memory than the BIOS knows about. In this case **DBOS** uses the portion specified by the given memory range in addition to that reported by interrupt 15.

For example, if interrupt 15 reports 16 megabytes of memory in a 32 megabyte machine then

```
DBOS 100000 2000000
```

will make all of the memory available as user page space, whilst

```
DBOS 200000 380000
```

will only use extended memory between 2 and 3½ megabytes. The values

```
DBOS 1000000 2000000
```

would make **DBOS** use the second 16 megabyte memory block in addition to that reported by interrupt 15. The virtual addressing capability of **DBOS** makes this transparent to **FTN77** programs.

This method does rely on the fact that you know how much memory the BIOS thinks it has. This information is usually available - simply invoking **DBOS** without any parameters will give a report on the amount of memory available via interrupt 15.

Memory management

FTN77 programs can make use of ordinary DOS memory at addresses beneath 640K and extended memory at addresses above 1 Mbyte. Since **DBOS** uses the hardware paging facilities provided in the 32-bit Intel chip, this memory can be used at any address where it is needed. Memory is controlled in pages of size 4096 bytes. Each page of memory starts on a 4096 byte boundary.

When a program starts execution all the free space from the DOS area and the extended memory is divided into pages. This pool of pages is then used to provide virtual memory where the program needs it. By default, **DBOS** will use all the

extended memory it can find for pages, although it is possible to tell DBOS to use less memory, so as to accomodate other uses of extended memory (see page 295).

If DBOS can find enough extended memory it will relocate most of itself above 640K so that only about 28K bytes of program remain resident in real mode. This is important, because it leaves more real mode space available for non-FTN77 programs. Although DBOS can be used with systems containing no extended memory, it is recommended that at least 256K bytes of extended memory be available.

Configuring DBOS

DBOS can be configured using the CONFIGDB utility. The DBOS_SET and DBOS_RESET commands may also be used to set various switches in DBOS once DBOS is loaded.

The CONFIGDB utility

CONFIGDB is a menu driven program which creates or modifies the file DBOS.CFG in your DBOS directory. This file is read once by DBOS, when DBOS is first loaded. After a successful reconfiguration, CONFIGDB will reboot your machine to activate the new parameter settings. Some users may find it convenient to save several versions of DBOS.CFG and copy the required one to the correct place just before loading DBOS. This avoids the need to use CONFIGDB each time the configuration parameters are to be changed.

If, for any reason, the DBOS.CFG file becomes corrupt, it might become impossible to load DBOS in order to run CONFIGDB to correct the situation! If this should happen, simply erase DBOS.CFG to return to the default options (it will be necessary to remove the read-only flag by issuing the MS-DOS ATTRIB command).

The following parameters are controlled by CONFIGDB:

☐ **Disk swapping**

DBOS can be instructed to use disk space as a swap area for programs which are too large to fit into memory. If you select this feature you will be asked to choose between swapping to an MS-DOS file or to a whole hard disk partition.

The latter method is more efficient, but requires a dedicated partition, and means that *all existing data on the disk partition will be destroyed*.

If you decide to use an MS-DOS file as a swap area, you will be asked to specify its size. This will be rounded down to a multiple of 128K bytes, and the file will be created of the required length. For best performance, it is important that this

file should not be fragmented. The use of a disk compression tool may be beneficial.

Note that the MS-DOS swap file must not be altered while DBOS is configured to make use of it.

If you decide to stop using a disk partition as a paging area, you should reformat the partition *after* using CONFIGDB to cancel its use.

□ **High resolution graphics**

A default high resolution graphics mode can be defined to be used by the routines HIGH_RESOLUTION_GRAPHICS_MODE@ and GET_GRAPHICS_RESOLUTION@ (but note that these routines have now been superseded by other more general routines). This may simply be set to standard VGA or EGA, but it gives the opportunity to exploit other extended modes available with some graphics cards. Some particular graphics cards are listed, but it is possible to provide information on the interrupts required and the resolution provided by any particular card, so the list is only for convenience, and does not constitute a limitation on the cards which can be driven.

□ **Run-time error actions**

Whether or not a machine register dump and a routine traceback is printed when a program aborts outside of the debugger can be controlled.

□ **Screen display**

The colours for normal text, window headings, highlighted text, debugger dialog and error message text within the FTN77 system can be configured.

□ **Miscellaneous features**

DBOS can make use of a technique which avoids the need to reopen the file DBOS.LIB each time an FTN77 program is executed. This can result in an improved response when a program is run, especially where slow hard disks are in use. However, this technique uses an undocumented feature of MS-DOS, which might therefore be unsupported in some future version or in some unusual software environment. It is suggested that this switch be set ON unless you are experiencing difficulties which you think may be related to the use of this feature. This is currently the only feature related to the “Miscellaneous” option on the CONFIGDB main menu.

If you decide to stop using a disk partition as a paging area you should reformat the partition after using CONFIGDB to cancel its use.

The DBOS_SET and DBOS_RESET commands

```
DBOS_SET <switch name>
DBOS_RESET <switch name>
```

These two commands are used to set or reset various switches in DBOS. Switch settings remain in effect until overwritten by another `DBOS_SET` (`RESET`) command, or until the end of the current DBOS session. Switch settings are not recorded on disk (c.f. `CONFIGDB`). Typically these commands may be included in your `AUTOEXEC.BAT` file.

The following switches have been defined:

TRACEBACK

If `SET`, this switch forces a traceback when a program fails outside of the debugger. Usually you would want this switch `SET`, however if you are working with assembly code instructions for example, and are only interested in the register dump, it may be useful to `RESET` this switch. The default value of this switch may be selected by using `CONFIGDB`.

HEXDUMP

If `SET`, this switch forces a register dump when a program fails outside of the debugger. Users who are not interested in the contents of these registers may prefer to `RESET` this switch. The default value of this switch may be selected using `CONFIGDB`.

PAGING

In the first instance, paging to disk is enabled by configuring DBOS using the `CONFIGDB` utility. Once paging has been enabled in this way, it can be temporarily disabled (and later re-enabled) by using this `PAGING` switch. If you are running applications under `DESQview`, paging is enabled by using the DBOS command line option `/PAGE` in addition to configuring DBOS using `CONFIGDB`.

QUIT_ON_ERROR

This switch, which is `SET` by default, causes programs which fail (including the compiler) to simulate a control break so that a batch file is interrupted without the need to explicitly test the return code. If the switch is `RESET`, then programs which fail will return a non-zero error code which may be tested using `'IF ERRORLEVEL'` in DOS.

The paging algorithm

The following description of the DBOS paging algorithm is not guaranteed to remain unchanged in all details in future versions of the software, although the overall mechanism will not change.

When a program starts to execute it begins to use up the pages of memory available from the pool of memory beneath 640K and above 1 megabyte. Memory can be used for any of the following purposes:

- ❑ Usually some extended memory will have been used by DBOS on startup to relocate part of itself outside the real mode address space.
- ❑ Memory is used for the program stack. This holds all dynamic variables and arrays (those which have not been saved) and various temporary variables and return links.
- ❑ Memory is used by uninitialised common blocks. Unless the /UNDEF option has been used, such common blocks are allocated page by page as they are used.
- ❑ Code inside the system library or from user-defined dynamic link libraries is paged in from disk as needed. This also consumes pages from the pool.
- ❑ A few pages are used for internal purposes. For example, pages are needed by the paging hardware itself to hold the page tables.

Extended memory pages are used preferentially for system library code because it is sometimes possible for DBOS to reuse such code from one program run to another without reloading it. If paging to disk is enabled, then most pages are candidates for being swapped out. If paging to disk is not employed, then only the system library code and unmodified portions of other memory-mapped MS-DOS files (chiefly the code of dynamic link libraries) can be swapped, since a copy of the information is already on disk ready to be reloaded when required. The above algorithm has a number of potentially interesting consequences for the user:

- Programs designed to operate on a range of problems with varying memory requirements can be written with one-dimensional arrays dimensioned to cater for the largest conceivable case. Providing the arrays in question are in dynamic local storage or uninitialised common, and provided also that the /UNDEF option is not used, only those portions of the arrays which are actually referenced will require physical memory.

Such programs can be made robust by using the subroutine GET_MEMORY_INFO@ (see below) to determine the actual memory available. Note that this technique is much less effective with multidimensional arrays, because data will be scattered in memory according to the standard Fortran memory organisation (first subscript varies fastest).

- There is almost no limit to the size of program that can be run if disk paging is used. The performance of programs will degrade gradually as the available memory is reduced.
- Very large suites of code which have traditionally been run using overlaying can reasonably be linked as a monolith and run with RUN77. The parts of the program which are not used will soon get paged out of memory.

- In a system containing a very large number of small subroutines, many of which will be rarely used, it may be worth specifying routines which use each other in adjacent `LOAD` commands in `LINK77`. This will tend to reduce the number of pages needed to hold the program.

Writing programs within memory capacity

If you are writing software which will be run on a range of 32-bit Intel hardware with differing amounts of extended memory available it is useful to have a way to avoid running out of memory and producing the *'Page memory exhausted'* error message. DBOS keeps a count of pages of real memory and of pages of disk space available. If disk paging is not enabled DBOS will let a program run out of memory completely, generating the *'Page memory exhausted'* message.

If disk paging has been enabled, however, DBOS produces the error message *'Down to page reserve'* while there are still enough pages remaining for the debugger to move in to store.

It is possible to alter the threshold at which the *'Down to page reserve'* message is produced, and it is also possible to trap this error using `SET_TRAP@` to perform some form of error recovery. The following program illustrates one useful way to combine these facilities. This program uses page swapping for debug and/or error recovery by forcing DBOS to fault only when the real memory has filled:

```

    INTEGER*4 TOTAL_DOS_PAGES,TOTAL_EXTENDED_PAGES,
    + REMAINING_DOS_PAGES,REMAINING_EXTENDED_PAGES,
    + TOTAL_DISK_SWAP_PAGES,REMAINING_DISK_SWAP_PAGES,
    + TOTAL_PAGE_TURNS
    INTEGER*4 JUNK
    EXTERNAL MY_ERROR_HANDLER

C
C  NOTE THAT THE FOLLOWING TWO SUBROUTINE CALLS COULD
C  BE REPLACED WITH ONE CALL TO TRAP_ON_PAGE_TURN@
C
    CALL GET_MEMORY_INFO@(TOTAL_DOS_PAGES,
    + TOTAL_EXTENDED_PAGES,REMAINING_DOS_PAGES,
    + REMAINING_EXTENDED_PAGES,TOTAL_DISK_SWAP_PAGES,
    + REMAINING_DISK_SWAP_PAGES,TOTAL_PAGE_TURNS)
C  FAULT WHEN ALL MAIN MEMORY IS IN USE (I.E. TOTAL
C  EQUALS DISK PAGE SPACE)
    CALL SET_PAGES_RESERVE@(TOTAL_DISK_SWAP_PAGES)
C
C  TRAP THE ERROR 'DOWN TO PAGE RESERVE'
```

```
C
CALL SET_TRAP@(MY_ERROR_HANDLER,JUNK,5)
CALL REST_OF_PROGRAM
END
INTERRUPT SUBROUTINE MY_ERROR_HANDLER
.
.
```

If you decide to use `GET_MEMORY_INFO@` to determine the memory available to you, you should be aware that the exact amounts of memory needed to run your program may vary for several reasons. For example, if a small routine happens to straddle a page boundary, then when it is used both pages will be required. Furthermore, a program may call different numbers of library routines depending on its data. However, even a small disk paging file can cushion you from these effects when calculating whether you have enough room to run your program.

One possibility for `GET_MEMORY_INFO@` is to use it to determine if your program will run without requiring page turns, and to print a suitable warning if it will require paging to disk - but then continue execution at the reduced speed.

Assembler instructions and the execution environment

When a 32-bit Intel chip is running MS-DOS it operates in 'real mode'. In this mode the chip emulates an 8086 chip with the addition of a few extra instructions and, of course, a much improved performance.

Real mode offers a 1 Megabyte address space composed of 64K byte segments and no protection of the operating system. In real mode, RAM at addresses above 1 Megabyte is unaddressable. The DBOS system stays resident above DOS and provides services via an interrupt (78 hex). The main service provided is to switch a program from real to protected mode. All FTN77 programs (including the compiler and all ancillary software) perform such a switch as the first instruction. The rest of a program executes in 32-bit protected mode with the paging hardware turned on and `CPL=3`. DBOS provides the tables known as the GDT, LDT, and IDT, which control protected mode operations.

Programs run with `CS`, `DS`, `ES`, and `SS` pointing to segments which overlay each other and offer an almost 2 Gbyte virtual address space. Observe that this implies that negative addresses are illegal. At the top of this space is an area of virtual memory reserved for the system library. Part of this space is write-protected and contains information which is demand-paged from the `DBOS.LIB` system file. Beneath this area is the system stack (pointed to by `ESP`), which is of a **BIG** descending type (as defined by the GDT entry for `SS`).

The user's program, which has normally been loaded as a .EXE file, lies at the bottom of the address space. Regardless of where the program has been loaded by MS-DOS, DBOS arranges the segment offsets so that the first location of the program is location 0. The .EXE files do not contain the space for common blocks (unless they have been initialised in a BLOCK DATA subprogram) or the system stack.

When DBOS starts a program (in response to the initial INT 78) it obtains all the unused memory below 640K by the appropriate DOS call, and pools this with memory residing above 1 Megabyte to provide pages for use as required by the program as it executes in protected mode. Uninitialised common and stack areas, for example, will be provided from this pool.

Since virtual memory hardware is used, user programs are not sensitive to the memory layout, which may vary from machine to machine, but only to the total amount of memory available.

Programs communicate with DBOS by the use of protected mode INT 78 instructions (which generate General Protection Exceptions into DBOS) followed by an identification byte indicating the service required. This combination is usually represented by a pseudo-op recognised by the FTN77 in-line assembler feature. Thus, for example, when a program wishes to terminate it issues:

```
INT      Z'78'          ; (Hex)
DB       0
```

which can be coded as:

```
SVC      0
```

Users should not need to use many of these SVC calls (which number about 50 in all), since the services which they offer have been packaged as callable routines. For example SVC/0 is available by calling EXIT or by executing the end statement in the main program. Most SVC's communicate with DBOS via the general registers. The small number of SVC's which may be of general use are described later in this chapter.

When a program is linked using the FTN77 linker, LINK77, or the /LGO compiler option there is no system library to be loaded. All system library routines (plus most of the debugger, linker, and compiler routines) are contained in the DBOS.LIB system file which, as explained above, is paged on demand into the system area above the stack.

The linker plants calls to these routines as calls to locations containing a byte DB(hex) which acts as an identifier, followed by a 1-byte name length followed by the name of the routine in question. In order to make the call illegal 80000000(hex) is added to the address (so as to make it negative). DBOS recognises this construct, follows the pointer and looks up the routine in its map of the system library. The call is then altered and re-executed.

This dynamic linking is very efficient and results in much smaller .EXE files than would otherwise be possible. If DBOS is unable to find a routine to satisfy an otherwise valid dynamic link it raises an error and reports that a call to a missing routine has been made. This illustrates the fact that not all General Protection Faults result in program errors, and of those that do, not all are reported explicitly as General Protection Faults.

If a program is run using RUN77, /LGO, /BREAK or /DBREAK, then DBOS handles most program failures by passing control to the source level debugger. This is inappropriate when debugging assembler and programs should be run as .EXE files without using RUN77. Program failures are then reported as register dumps. For example:

```
Coprocessor fault (status =B882, instruction address = 00000051)
Denormalised operand at User/00000054
Flags=00010246 (EQ No carry Odd parity DF = forward)
EAX%=00010000 EBX%=0000006F ECX%=00000001 EDX%=00000000
EBP%=7FCFFFF6 ESI%=7FCFFFF6 EDI%=7FCFFFC0 ESP%=7FCFFFF6
ST(0)=0.20000000000000003E-41
00000054) FSTP DS:[EBX%-10]
```

The display contains an explanation of the error at the assembler level, a dump of all the registers, and a print of the next instruction to be executed. All values are in hex except for the contents of the co-processor stack.

The fault address (in this case User/54) is normally in the user space, although it is possible to generate faults inside DBOS in which case the fault address would take the form Os/<hex no>. The debugging of programs at this level is described in chapter 7.

Using assembler instructions to call DOS and BIOS

Real mode assembler programmers are familiar with calling DOS and BIOS routines by loading information into the (16-bit) registers and issuing the appropriate INT instruction. Many DOS and BIOS facilities have been made available via the library routines supplied with the compiler. For example, it is recommended that all file access operations are performed by library routine call rather than DOS calls.

The usual way to access DOS or BIOS directly is via the subroutine REAL_MODE_INTERRUPT@ (for details of this and other routines in this section see the *FTN77 Library Reference* manual). However, to cater for special requirements, and for compatibility with earlier versions of FTN77, SVC/3 has been provided. This SVC should be followed by one byte defining the interrupt required. When SVC/3 is executed DBOS will switch to real mode and issue the corresponding

INT instruction with the 16-bit registers filled with the bottom 16 bits of the user's register set. For example, in order to read the printer status word using BIOS you could execute:

```

INTEGER*2 STATUS
CODE
  MOVB    AH%,2      ;Status code
  SVC     3          ;Ask DBOS to perform operation
  DB      z'17'      ;Printer interrupt
  MOVB    AH%,0      ;Zero extend result
  MOVH    STATUS,AX% ;Store the result away
EDOC

```

This is adequate for most purposes, but it does not cater for operations which require pointers to memory. The problem is that your Fortran code is running in a paged environment and bears no simple relationship to real mode memory. To cater for this problem DBOS provides a special segment which overlaps a piece of real mode memory. The DS and ES registers will be set to point to this segment every time SVC/3 is executed. The segment is defined to be 512 bytes in length, and its descriptor can be loaded into FS by a call to DOSCOM@. For example, this is the code (somewhat simplified) for the COUA built in routine which prints a string on the standard output:

```

SUBROUTINE COUA(C)
CHARACTER*(*) C
INTEGER*4 L
EXTERNAL DOSCOM@
C
C   OBTAIN THE LENGTH OF STRING TO PRINT
C
  L=LEN(C)
  IF(L.LE.0)RETURN
CODE
  CALL    DOSCOM@      ;Load FS with pointer
+         ;    to DOSCOM seg
  MOV     ECX%,L
  MOV     ESI%,C        ;Point to string
  MOV     EDI%,0        ;Point to start of DOSCOM seg
1  LODSB                ;Load a character
  FS                      ;FS: prefix
  MOV     [EDI%],AL%    ;Put it into DOSCOM segment
  INC     EDI%
  DEC     ECX%
  JG      $1            ;Loop round until done
  MOVH    DX%,0
  MOV     ECX%,L        ;Amount to write

```

```

MOVH    AX%,Z'4000' ;DOS write function call
PUSH    EBX%        ;Must save EBX
MOVH    BX%,1        ;Handle 1 is standard output
SVC     3            ;Ask for real mode interrupt
DB      Z'21'        ;DOS call
POP     EBX%         ;Restore EBX or program
+                               ;    will go wild
EDOC
END

```

The regenerative screen buffer is also available on a similar basis as a separate protected mode segment. This segment starts at real mode address A0000, so depending on the mode you are using with your video adaptor you may need to offset into the array. Writing into this segment will, of course update the contents of your screen at high speed. To load FS with the appropriate selector call SCREENSEG@.

DBOS memory map

The following table shows memory areas which should not be allocated to link libraries as they are reserved for use by software marketed by Salford Software Ltd.

Address	Usage
41000000 - 41FFFFFF	SGKS, LIBCGKS.LIB
43000000 - 43FFFFFF	SGKS, LIBFGKS.LIB
4C000000 - 4CFFFFFF	Pascal dynamic link library.
50000000 - 50FFFFFF	NAG FTN90 maths library.
50000000 - 51FFFFFF	Pascal heap.
51000000 - 51FFFFFF	NAG FTN77 maths library.
52000000 - 52FFFFFF	GKS.
53000000 - 53FFFFFF	IMSL for FTN90
54000000 - 54FFFFFF	ISML for FTN77
57000000 - 57FFFFFF	FTN90 run time system.
60000000 - 7ECFFFFFF	DBOS.
7ED00000 - 7FFFFFFF	DBOS default storage heap.

Running DBOS applications under Windows (Win16)

Introduction

The DBOS system includes the Windows interface, known as WDBOS. WDBOS is necessary only if you intend to use DBOS, or run a DBOS program, under Microsoft Windows 3.1 or Windows 95. WDBOS provides the interface to Windows memory management functions and other services that are vital to the correct operation of DBOS in this environment.

Unlike some other multi-tasking environments, Windows does not offer DOS programs access to the Virtual Control Program Interface (VCPI) for switching to protected mode. One of the functions of WDBOS is to provide a subset of these missing functions. Using WDBOS it is possible to run one or more copies of DBOS concurrently and to switch between DBOS programs and other Windows applications within a Windows session.

Installing WDBOS

WDBOS is a virtual device driver which is in the file WDBOS.386. This file is copied into the DBOS system directory on your hard disk during the installation procedure. To use WDBOS, Windows 3.1 must be informed that WDBOS.386 is to be loaded as a device driver. This information should be included in the Windows SYSTEM.INI file. The installation procedure will update your SYSTEM.INI file automatically if you give the necessary confirmation when prompted. Failing this, you can update the file manually as follows:

- ☐ Change to your WINDOWS directory.

- Edit the **SYSTEM.INI** file using a text editor that does not append Ctrl-Z to the file. Find the “[386enh]” section. On a fresh line, after the last “device=” directive, enter:

```
device=c:\dbos.dir\wdbos.386
```

or the equivalent for your configuration (note there is no asterisk following the “=”).

A typical extract from a **SYSTEM.INI** file is shown below:

```
. . .
device=*biosxlat
device=*vdc
device=*vmcpd
device=*combuff
device=*cdpscsi
device=c:\dbos.dir\wdbos.386
local=CON
FileSysChange=off
. . .
```

Once this procedure has been carried out, **DBOS** may be used in a DOS window in Windows 95 or Windows 3.1 “enhanced” mode.

Windows modes

Windows 3.1 provides two modes of operation: *standard* and *enhanced*. These modes allow you to run one or more 16-bit application programs, each in a DOS window (commonly called a “DOS Box”). To the **DBOS** user, only the Windows 3.1 enhanced mode of operation is of interest, as this is the mode that offers full access to the functionality of the 32-bit Intel chips. Windows 95 always uses “enhanced mode”.

The command:

```
WIN/3
```

starts Windows in enhanced mode, where DOS programs execute in virtual mode and Windows applications execute in 16-bit or 32-bit protected mode.

If you start Windows with the command:

```
WIN
```

(i.e. with no parameter) then Windows determines the mode that will be used. On a 32-bit machine the **WIN** command will use enhanced mode if sufficient extended

memory is available. If there is insufficient extended memory the WIN command will enter standard mode without reporting the fact to you.

For this reason, DBOS must not be loaded before Windows is started. If DBOS has been loaded, then it should be unloaded by typing:

```
KILL_DBOS
```

before Windows is started.

Running programs in a DOS box

To invoke a DOS session from within Windows 3.1, simply double click on the “MS-DOS Prompt” icon in the Program Manager (Main) menu. Under Windows 95 click on the Start icon at the foot of the screen and move the mouse to Programs. Then select “MS-DOS Prompt”. DBOS may be then invoked by typing

```
DBOS
```

as you would do under DOS. Please note that the DBOS option /DISK_CACHE is not available when using DBOS within a DOS Box in Windows.

A DOS session may be windowed or may occupy the full screen. Switching between full screen or windowed operation is done by pressing the Alt-Enter key combination.

You can switch between your DOS Box and Windows by pressing Alt-Tab.

If you want to remove DBOS, and exit from a DOS box, type

```
KILL_DBOS  
EXIT
```

If you don't use KILL_DBOS before typing EXIT, you may be presented with a small window asking you to use your “pop up program” and type Ctrl-C to return to Windows. If this happens, ignore the message and type Ctrl-C. This message is normal and does not indicate any fault. Typing EXIT will terminate that DOS session and DBOS will be removed from memory. The memory used by DBOS during the DOS session will be released. In Windows 95 you can exit by clicking on the X-box in the top right hand corner of a windowed DOS box.

As an alternative to the above procedure you may prefer to create a .BAT file containing a call to DBOS (and HOTKEY77 with HELP77 if you prefer) followed by a call to COMMAND.COM (or perhaps a call to your application).

Under Windows 3.1, you may then use the Windows PIF Editor to create a corresponding .PIF file. The Windows Program Manager can then be used to attach

this .PIF file to an icon in order to run DBOS from the Program Manager. This will emulate the MS-DOS prompt icon.

Under Windows 95, once a batch file has been created it is possible to set up a short cut to the batch file in either of the following two ways.

- Click the right mouse button on the desktop to bring up a popup menu. Select the **New** option followed by **Shortcut**. Now either enter the full path and filename of the batch file or use the **Browse** option to find the file. Once you have selected the file you will be prompted to select a suitable icon. This icon will then appear on the desktop and when you double click on the icon the batch file will be called.
- Alternatively, to create a **Start** program group entry, click with the right mouse button on the **Start** icon and select **Open** from the menu. Now double click on the **Programs** group so that it opens. Next double click on **Your Computer** icon and select the batch file and using the right mouse button, drag it into the **Programs** area. Select **Create Shortcut**, from the menu that appears. You will now be able to start the batch file from **Start** and **Programs**.

Switching back to Windows

The key combination **Alt-Tab** will invoke the Windows task switcher. Successive depressions of the **Tab** key, keeping the **Alt** key depressed, will cycle through the currently running applications.

Releasing the keys will select the currently visible application. The DOS session will be shown iconised at the bottom of the screen. Double clicking on this icon or using the task switcher will enable you to reselect your original DOS session. Under Windows 95 you can use the Taskbar bar the is normally at the foot of the screen.

It is possible to have multiple DOS sessions and therefore multiple DBOS sessions. Each one of these DOS sessions may be regarded as running on a different machine, or “Virtual Machine” in Windows terminology. DBOS must be invoked separately for each DOS session in which it is required. The exception to this rule is the case when you have task-switched back to Windows leaving the DOS session active and have subsequently task-switched back to the same DOS session.

Achieving the above is much easier if you are using windowed DOS sessions. Task switching may then be carried out by clicking on the window of the application or DOS session that you wish to select.

WDBOS version number

A simple Windows utility application, WDBOSVER.EXE, is provided. This utility can extract the version number of the WDBOS.386 device driver present on your system. WDBOSVER should only be run under Windows in enhanced mode.

Plotter Interfacing (DOS)

The plotter

The comments below relate to the HP 7550a plotter. If you have another type of plotter you are advised to read the manuals supplied with your plotter to see how it should be interfaced to the PC.

Cabling requirements

The correct plotter cable is essential for the successful use of the plotter. The cable must be suitable for hardwire handshaking and cables of the 'straight thru' type or cables designed to be used with packages that only support XON/XOFF handshaking should not be used. Figure 25-1 shows the correct pin outs for such a cable. Typical errors caused by use of the incorrect cable are: the DOS 'Access Denied' at the PC end and 'I/O buffer overflow' at the plotter end.

Panel settings

The PC and the plotter must be set to the same baud rate. Other typical settings are:

BYPASS	OFF
HANDSHAKING MODE	HARDWIRE DIRECT
DATA FLOW	REMOTE STANDALONE

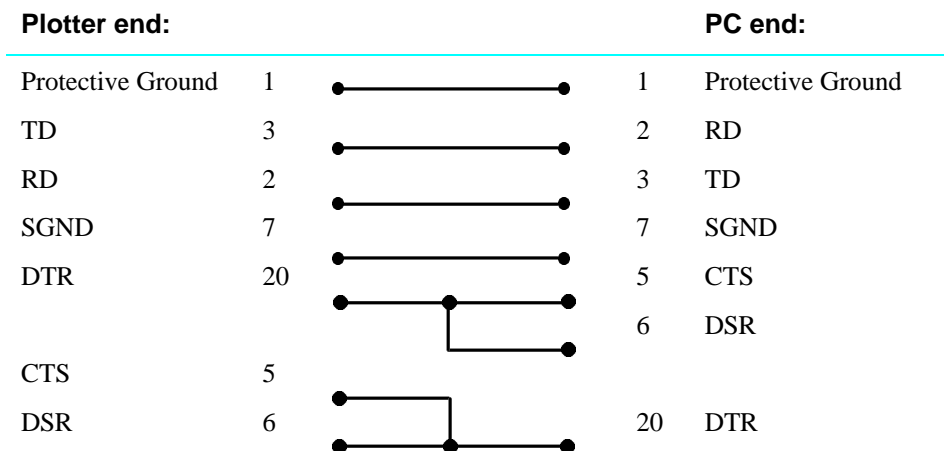


Figure 25-1 Plotter cable pin-out

Plotting plot files

There are three ways to plot the HPGL files produced by FTN77.

- 1) Use the **COPY** command. If the plotter is attached to **COM1** and the file is called **PLOT.PLT**, then type:

```
COPY PLOT.PLT COM1
```

- 2) Use the **PRINT** command. Type:

```
PRINT PLOT.PLT /D:COM1
```

- 3) Redirect **LPT1** to **COM1**. Type:

```
MODE LPT1:=COM1 Redirects LPT1 to COM1
PRINT PLOT.PLT Plots PLOT.PLT
```

The redirection need only be done for the duration of the **DOS** session. All subsequent prints will automatically be channeled through **COM1**. To terminate redirection type:

```
MODE LPT1:80,6 or similar
```

COM1 may 'time out' whilst paper is being fed or the pen changed. If this happens add a 'p' to the initial mode command that sets the **COM1** parameters e.g.

```
MODE COM1:9600,n,8,1,p
```

This will cause a continuous polling of **COM1** should such a 'time out' condition occur. The 'time out' loop may be terminated with **Ctrl-Break**.

Calling real mode libraries and programs (DOS)

Introduction

This chapter describes how you can adapt existing real mode libraries and programs for use with programs compiled with FTN77. If you are not familiar with the various execution modes provided by a 32-bit Intel CPU chip you will find it helpful to read the next section before attempting to follow the description starting on page 328.

The library subroutines provided by FTN77 and referenced here are described in detail in the *FTN77 Library Reference*.

Real and protected modes

A 32-bit Intel CPU differs in several fundamental ways from the original 8088 CPU around which MS-DOS was designed. The most obvious and, for FTN77, most vital characteristic is the ability to manipulate 32-bit data and to use 32 bits to address data, thus making up to four gigabytes of memory theoretically available. 32-bit chips also gain in speed of data access as they manipulate information in chunks four times bigger than those of the IBM PC's 8088 processor, and twice the size of the IBM PC/AT 80286's chunks.

Another, less immediately obvious, distinction is the 32-bit chip memory-management facilities. These facilities break down into three distinct modes of operation. The first is *real mode*, which is provided to guarantee compatibility with existing applications. The processor emulates a fast 8088 or 8086. When operating in real mode, a 32-bit chip cannot address more than 1Mbyte of memory, just like the 8088/86.

The 80286 and all 32-bit chips have a *protected mode* of operation. 80286 protected mode allows up to 16 megabytes of data to be addressed, whereas on 32-bit Intel chips the protected mode makes four gigabytes of memory addressable. Both protected modes also support virtual memory techniques, whereby data can be swapped to and from disk when real memory is full. This allows several applications to run in memory concurrently, each protected from the others' anti-social behaviour by the chip's memory management. The 80286 chip will operate in either protected mode or in real mode, but a change of mode requires the system to be rebooted, thus limiting its usefulness in many applications.

32-bit Intel chips have a feature not available with the 80286 chip: *virtual 8086 mode*. Here the processor segments memory into 1Mbyte chunks, complete with 640K limit, each of which appears to a DOS application to be an independent machine. Unaltered, multiple DOS applications can run in one box without interfering with each other.

Unfortunately, with a 32-bit Intel chip, DOS applications cannot use protected mode directly. There are two approaches to making the power and addressing capability available to programs:

- 1) Provide a new operating system, such as OS/2 or Windows 95/NT, and write applications which can exploit it.
- 2) Make use of a "DOS Extender", such as our DBOS package.

Using DBOS, programs can make use of 32-bit Intel protected mode and switch readily in and out of real mode. Thus, FTN77 programs can co-exist with DOS applications. DBOS allows full use of 32-bit addressing and uses the paging feature of the 32-bit Intel CPU so that programs can address up to 4 gigabytes of memory. When an FTN77 program has been compiled and link-loaded, the address space which is used for, say, a common block, might be fragmented in physical memory, because of the way the paging algorithm works.

Rules for calling real mode from protected mode

This process will seem at first like a 'kludge' which, of course, it is. We stress that there is no standard way provided by the 32-bit Intel chip to achieve real and protected mode inter communication. Real mode and protected mode code may be loaded at the same time into physical memory. The mode in which the 32-bit Intel CPU is operating can be switched under the control of the DBOS system so that it is possible to 'call' real mode code from a protected mode program.

However, data which is accessible in 32-bit Intel protected mode (such as an array), may be fragmented in physical memory. If real mode code needs to access data which

is in a protected mode program, or vice versa, the data must be copied in physical memory. It should always be kept in mind that this process of mode switching and data copying has an execution time overhead associated with it.

Calling real-mode libraries

In order to use a real-mode library from an FTN77 program, you will have to do the following:

- 1) Prepare a standard MS-DOS executable (.EXE) file, by compiling and linking a main program and library routines using your chosen real mode compiler(s) (and linker if provided), such as Professional Fortran, F77L, MASM, etc. and the MS-DOS linker, LINK. As you will see from the simple example on page 330, the main program should contain all of the following:
 - a) a common block which will be used to transfer data between the real and protected mode programs;
 - b) if more than one real mode 'service' is to be controlled by the program, a Fortran computed GOTO statement, or its equivalent, which will allow the various operations in the real mode program to be controlled from a variable whose value is set by the protected mode program which calls it;
 - c) at least one call to the subroutine FTN77WT (supplied on the FTN77 release disk in source and object forms), which is used to initialise the real mode/protected mode interface and subsequently to return control to the protected mode program each time that the real mode program completes any stage of execution. Note that this routine has a number of entry points as different real mode compilers use different calling conventions. Details of these entry points are as follows:

Real Mode Compiler	Entry point name
Lahey F77L	FTN77WTL
Prospero Fortran	FTN77WTP
Professional Fortran	FTN77WT
Microsoft Fortran	FTN77WTM
Turbo C	FTN77WTC

- ☐ The single argument to FTN77WT may be of any type.
 - ☐ Any of the above entry points may be used with real mode assembler code compiled with MASM. Which entry point will depend on the calling conventions you have chosen.
- 2) Incorporate the following into your FTN77 protected mode program:

- a) A common block which will be used to transfer data between the real and protected mode programs. This common block should be the same size in bytes as that described in 1(a), above.

Note: The two common blocks have the same definition but, because of the limitations of the 32-bit Intel CPU chip described on page 327, they do not represent the same area of physical memory. It is necessary to copy data between the two physical common block locations.
- b) Calls to several special system subroutines in the FTN77 library which initialise the mechanism and transfer data between the common blocks in the real and protected mode programs.
- 3) It is possible to use the FTN77 debugger while programming in this way, but it is not possible to use any real mode facility such as Microsoft's CODEVIEW or Lahey's SOLD.
- 4) The real mode program should not terminate with STOP, END or CALL EXIT. Termination of execution should always be in the protected mode program.
- 5) Both the real and protected mode programs can perform input/output, but the same file should not be open simultaneously in both programs.

The example which follows will clarify the preceding explanation.

The example programs listed here are provided on the distribution diskette in the DEMO directory. The protected mode program is provided in source and executable form (PPROG.FOR and PPROG.EXE). The real mode program is provided in source form (RPROG.FOR). An example output file, PPROG.OUT, is also on the diskette.

Protected mode program (FTN77)

```

C   Declare common block etc.
C
      INTEGER*4 K,ICOMSIZE
      PARAMETER(IX=10,ICOMSIZE= IX*4 + 8)
      INTEGER*2 X(IX),Y(IX),MAX,MIN
      COMMON K,X,Y,MAX,MIN
C
C   Start with some calculations using one of your own
C   FTN77-compiled subroutines:
C
      CALL CALC(X,Y,IX)
C
C   The call to LOAD_REAL_MODE_LIBRARY@ loads the real
C   mode program and initialises the calling mechanism.
C
C   The first executable

```



```

C   statement in the real mode program should be a call to
C   the subroutine FTN77WT, which returns control to this
C   program at the statement following this call.
C
C       CALL LOAD_REAL_MODE_LIBRARY@('C:\FTN77.DIR\DEMO\RPROG.EXE')
C       PRINT*
C       PRINT *, 'Returned to protected mode with K = ', K
C       K=1
C
C   The following call moves ICOMSIZE bytes of data TO the
C   real mode program's common block. Note that both
C   arguments must be INTEGER*4
C
C       CALL COPY_TO_REAL_MODE@(K, ICOMSIZE)
C
C   The real mode program is invoked using a value of 1 for
C   the flag, K
C
C       CALL REAL_MODE@
C
C   The results are copied back from the real mode program.
C   The following call moves ICOMSIZE bytes of data FROM the
C   real mode program's common block. Note that both
C   arguments must be INTEGER*4
C
C       CALL COPY_FROM_REAL_MODE@(K, ICOMSIZE)
C
C   And printed
C
C       PRINT*
C       PRINT*, 'Maximum and minimum values are', MAX, MIN
C
C   Now call real mode program to use a plotting routine
C
C       K=2
C       CALL COPY_TO_REAL_MODE@(K, ICOMSIZE)
C       CALL REAL_MODE@
C       CALL COPY_FROM_REAL_MODE@(K, ICOMSIZE)
C       PRINT*
C       PRINT*, 'Returned from real mode after plotting'
C       END
C       SUBROUTINE CALC(X,Y,IZ)
C       INTEGER*2 X(IZ),Y(IZ)
C       REAL RANDOM
C       DO I=1, IZ

```

```

      X(I)=I
      Y(I)=RANDOM()*32000.0
      END DO
      END

```

Real mode program

```

C   Declare common block.
C   This common block declaration should exactly match that
C   in the FTN77 program
C
      INTEGER*4 K
      PARAMETER(IX=10)
      INTEGER*2 X(IX),Y(IX),MIN,MAX
      COMMON K,X,Y,MAX,MIN
      PRINT*,'Real mode initialised'
C
C   This subroutine is used to return control to the
C   protected mode program each time this real mode
C   program has finished a stage of its execution.
C   Notice that all calls to subroutine REAL_MODE@
C   in the FTN77 program start execution from
C   the statement after this call.
C
1    CALL FTN77WTL(K)
      PRINT *, 'Real mode called with K = ',K
      GOTO(99,2,3),K+1
      PRINT*,'Invalid K value'
      GOTO 99
C
C   Data manipulation using a real mode subroutine
2    PRINT *, 'Real mode service 1 - max/min'
      CALL MM(Y,IX,MAX,MIN)
      GOTO 1
C
C   Plotting using a real-mode subroutine
C
3    PRINT *, 'Real mode service 2 - plotting'
      CALL HIST(X,Y,IX)
      GOTO 1
99   CONTINUE
      PRINT *, 'Real mode at end with K = ',K
      GOTO 1
      END
      SUBROUTINE MM(Y,IY,MX,MN)

```

```

        INTEGER*2 Y(IY),MX,MN
        MN=32767
        MX=0
        DO 1 I=1,IY
        MX=MAX(MX,Y(I))
        MN=MIN(MN,Y(I))
1       CONTINUE
        END
        SUBROUTINE HIST(X,Y,IX)
        INTEGER*2 X(IX),Y(IX)
        DO 1 I=1,IX
        IC=Y(I)/1000
        WRITE(*,100)X(I),(' ',J=1,IC)
1       CONTINUE
100     FORMAT(1X,I2,1X,32A)
        END

```

Real mode program (C example)

```

/*
  Declare common block equivalent.
  This common block declaration should exactly match
  that in the FTN77 program. This relies on the
  property that most compilers will put consecutively
  declared data contiguously in memory, all passed data
  may need to be placed in an array to ensure
  contiguous storage.
*/
#define IX 10
long K;
float X(IX),Y(IX);
extern void far FTN77WTQC(long far*);

...

puts("Real mode initialised");
K=100;
/*This subroutine is used to return control to the
protected mode program each time this real mode
program has finished a stage of its execution.
Notice that all calls to subroutine REAL_MODE@ in
the FTN77 program start execution from the statement
after this call. */
while (1)
{ FTN77WTQC(&K);

```

```

printf("Real mode called with K = %ld\n",K);
switch (K)
{   case 1 :
        /* Data manipulation using a real mode
        subroutine */
        puts("Real mode service 1 - sorting");
        sort(X,Y,IX);
        break;
    case 2 :
        /* Plotting using a real mode subroutine */
        puts("Real mode service 2 - plotting");
        plothp(X,Y,IX);
        break;
    case 0 :
        printf("Real mode at end with K = %ld",K);
        break;
    default :
        puts("Invalid K value");
        break;
}
}

```

Sample output

Real mode initialised

Returned to protected mode with K = 0

Real mode called with K = 1

Real mode service 1 - max/min

Maximum and minimum values are 25864 1410

Real mode called with K = 2

Real mode service 2 - plotting

```

1 *
2 *****
3 *****
4 *****
5 *****
6 ***
7 *****
8 ***
9 *****
10 *****

```

Returned from real mode after plotting

Notes:

- 1) The protected mode program contains the path name of the real mode library. This will need to be changed, and the program recompiled, if you have installed your compiler/demo programs differently.
- 2) The real mode program contains a call to the routine `FTN77WTL`. This is an entry point in the subroutine `FTN77WT` provided by `FTN77`. You may need to change this to the entry point appropriate for your real mode compiler before compiling this routine.
- 3) When you have compiled `RPROG.FOR` with your real mode compiler, it should be linked with `FTN77WT.OBJ`, plus your compiler libraries.
- 4) By comparison of the routines above with the sample output, it is possible to see the following standard sequence of events:
 - a) The call to `LOAD_REAL_MODE_LIBRARY@` from the protected mode program initialises the system by entering the real mode main program, executing the call to `FTN77WT` and returning immediately to the protected mode program.
 - b) Subsequent calls to `REAL_MODE@`, with appropriate values for the common variable `K`, enter the real mode program at the line following the call to `FTN77WT`, execute as appropriate and terminate with a call to `FTN77WT`. Execution is returned to the protected mode program.
 - c) Program execution is always terminated (by `STOP`, `END` etc.) in the protected mode program.

Calling real-mode drivers

Drivers are terminate-and-stay-resident programs (TSRs). Examples are those external drivers provided by `GSS`, and the `BTRIEVE` record manager program. Once loaded, the TSR remains ready to be activated by an interrupt from some other program or external source. In these cases, it may be possible to use a more straightforward method with which to call real mode code. The subroutines

```
ALLOCATE_REAL_MODE_MEMORY@  
COPY_TO_REAL_MODE1@  
COPY_FROM_REAL_MODE1@  
REAL_MODE_INTERRUPT@
```

are used as described below:

- 1) Install the real mode TSR.

- 2) The protected mode program should allocate real mode memory for itself using `ALLOCATE_REAL_MODE_MEMORY@`. For example, in order to allocate 3000 bytes of real memory, the protected mode program should use the following call:

```
INTEGER*4 I4PTR
INTEGER*2 IC
CALL ALLOCATE_REAL_MODE_MEMORY@(I4PTR,INTL(3000),IC)
```

The use of `INTL` ensures that the argument is type `INTEGER*4`. `IC` is a status code which is zero for successful allocation. `I4PTR` is a real mode address which can be used in the protected mode program.

If the memory requirement is greater than 30000 (thirty thousand) bytes, the `DBOS` command `COMSPACE` should be used to reserve an appropriate amount of real memory. For example, in order to reserve 40000 bytes for your protected mode application, use

```
COMSPACE D'40000'
```

- 3) The protected mode program can now use a data area of its own to prepare information for the real mode program. This information is then copied to the allocated area of real mode memory using `COPY_TO_REAL_MODE1@`. For example,

```
COMMON/PROTCD/DATA_AREA(1000)
INTEGER*2 DATA_AREA,IC
INTEGER*4 RMADDR
. . .
CALL ALLOCATE_REAL_MODE_MEMORY@(RMADDR,INTL(3000),IC)
IF(IC.NE.0)STOP'Failed to allocate real mode memory'
. . .
CALL COPY_TO_REAL_MODE1@(DATA_AREA,INTL(2000),RMADDR+1000)
```

This example allocates 3000 bytes of real memory, then copies data into this area starting at offset 1000.

- 4) The driver can then be called using an interrupt as follows:

```
CALL REAL_MODE_INTERRUPT@(REGISTERS,INTERRUPT)
```

- 5) Any result data can be copied back from the real mode driver as in the following example:

```
CALL COPY_FROM_REAL_MODE1@(DATA_AREA,INTL(2000),RMADDR)
```

Note: the arguments for `COPY_FROM_REAL_MODE1@` are in the same order as for `COPY_TO_REAL_MODE1@`.

Execution errors and IOSTAT values

All execution error messages consist of a message in English. These messages are listed below. Execution errors corresponding to input/output statements can be trapped by means of the `ERR=` and/or `IOSTAT=` keyword specifiers used with the input/output statements (see page 103). The value returned by `IOSTAT` in this case is n where n is the execution error number that appears in the table below. Users are advised to trap specific errors by means of `IOSTAT` rather than to continue execution regardless of the error that has been detected by the input/output system.

Notes:

- ☐ The `IOSTAT` value -1 indicates that an end-of-file condition has occurred.
- ☐ The positive values chosen for `IOSTAT` in this implementation of Fortran 77 will, in all probability, differ from those chosen in any other implementation for the same error conditions.

Error No.	Message
0	No error
1	Floating point arithmetic overflow
2	Integer arithmetic overflow
3	Argument to <code>CHAR</code> outside range 0 - 255
4	Character argument/function name of wrong length
5	Attempt to execute invalid assigned <code>GOTO</code>
6	Inconsistent call to routine
7	<code>DO</code> -loop has zero increment
8	User-specified range check error
9	Might be array bound error or corrupt program - rerun with checks

10	Lower substring expression > upper
11	Array subscript(s) out-of-bounds
12	Lower substring expression out-of-range
13	Illegal character assignment
14	Attempt to alter an actual argument that is either a constant or a DO variable
15	Attempt to access undefined argument to routine
16	Lower array bound > upper bound
17	Upper substring expression out-of-range
18	This routine has been entered recursively (/ANSI mode)
19	Actual array argument size smaller than dummy array argument size
20	Argument to SINH/COSH out of range
21	Zero raised to negative or zero power
22	Floating point division by zero
23	Floating point arithmetic underflow
24	This source has not been compiled with /PROFILE
25	Argument to EXP out-of-range
26	Argument to ASIN/ACOS out-of-range
27	Invalid floating point number
28	Negative argument to square root
29	Call to missing routine
30	Storage heap is corrupt
31	Floating point number too big for integer conversion
32	Second argument to MOD is zero
33	Both arguments to ATAN2/DATAN2 zero
34	Negative or zero argument to logarithm routine
35	Illegal argument to TAN routine
36	Negative number raised to non-integer power
37	Integer divide overflow
38	Illegal character assignment (R.H.S. overlaps L.H.S.)
39	Illegal window
40	No more windows available
41	Maximum number of breakpoints already set
42	This line number is not available as a breakpoint
43	Invalid command

44	Unable to open file
45	String not found
46	Routine not found or not compiled in check mode
47	Invalid expression
48	No more room for debugger information
49	Attempt to call a block data subprogram
50	Undefined input/output error
51	Format/data mismatch
52	Invalid character in field
53	Overflow detected by input/output routine (data out-of-range)
54	$m > w$ in <i>Iw.m</i> run-time format
55	$m > w$ in <i>Ow.m</i>
56	Unit has been closed by means other than a CLOSE statement
57	Attempt to read past end-of-file
58	Corrupt listing file
59	There is no repeatable edit descriptor in this format
60	Invalid external unit identifier
61	Invalid scale factor
62	Invalid or missing repeat count
63	Preconnected file comprises formatted records
64	Preconnected file comprises unformatted records
65	This command is not permitted from this window
66	File not in correct format
67	Character buffer too small
68	Field width exceeds direct access record size
69	Invalid record length (see documentation)
70	Logical input field is blank
71	H or apostrophe editing not allowed for input
72	Repeated formats nested too deep (>10)
73	Missing opening parenthesis in 'run-time' format
74	Invalid editing descriptor
75	A zero or signed repeat count is not allowed
76	Repeat count not allowed
77	Digit(s) expected

78	Decimal point missing
79	Missing separator
80	Invalid ACCESS specifier
81	Invalid combination of specifiers
82	ANSI - RECL is an invalid specifier
83	Label does not reference a format statement
84	Only BLANK may be changed for a file that exists for a given program
85	Repeated character constant must not extend past the end of a line
86	Character input/output list item is part of internal file
87	ENCODE/DECODE character count zero or negative
88	Internal file must not be constant or expression
89	Attempt to write past end of internal file
90	File access and properties are incompatible
91	Missing) from complex number
92	Invalid CLOSE statement
93	Missing (from complex number
94	Unit has neither been OPENed nor preconnected
95	Invalid direct access record number
96	Illegal operation (BACKSPACE/ENDFILE/REWIND) on a direct access file
97	Direct access record length too big
98	Invalid FILETYPE specifier
99	A function which performs I/O must not be referenced in a WRITE or PRINT statement
100	List-directed input/output is not allowed with direct access
101	Direct access is not allowed with an internal file
102	A formatted
103	Missing FILE specifier
104	File positioned at end-of-file
105	Invalid record length for existing direct access file
106	A valid record length must be specified if access is direct
107	STATUS=NEW must not be used with an existing file
108	Direct access record length mismatch
109	Brackets nested too deeply (>20)
110	Unformatted record is corrupt
111	Coprocessor invalid operation

112	Reference to undefined variable or array element (/UNDEF)
113	Insufficient allocatable storage
114	Emulator failure
115	Invalid hash table
116	Too many files open
117	Disk full
118	ANSI - exponent out-of-range (use Ew.dEe or Gw.dEe edit descriptors)
119	Down to page reserve
120	Reference to non-existent Weitek coprocessor
121	Too many registered traps
122	No high resolution graphics mode is available
123	Too many labels in debug macro file
124	This command is only allowed in a macro
125	A file of this name already exists
126	ANSI - invalid STATUS specifier
127	ANSI - invalid edit descriptor
128	File does not exist
129	Invalid attempt to use peripheral
130	Unformatted record too big
131	ANSI - octal/hexadecimal/binary input not permitted
132	Device type not known on this installation
133	Expression required
134	File already in use
135	Sign not at start of field in business editing descriptor
136	Business editing not allowed for input
137	Illegal operation after a BACKSPACE
138	Attempt to write to readonly file or inconsistent file access
139	You may not write to a file that is 'READONLY'
140	You cannot OPEN a directory
141	ANSI - invalid \$ in format descriptor
142	\$ editing not allowed for input
143	Incorrectly positioned \$ character in format descriptor
144	Illegal name in OPEN/CLOSE/INQUIRE statement
145	ANSI - the Aw edit descriptor must be used with an item of type CHARACTER

146	File path not found
147	Macro label not found
148	Reference to undefined variable or array element (/UNDEF)
149	Value returned by RECL= or NEXTREC= will cause overflow (use INTEGER*4 instead of INTEGER*2)
150	Count for ENCODE/DECODE must be in the range 1 to 32767
151	Invalid FORM specifier
152	Invalid STATUS specifier
153	Invalid BLANK specifier
154	Unpaired brackets
155	Error detected by user-specified device driver
156	Unexpected error in Fortran I/O system
157	Do-loop will never be executed (/DOCHECK)
158	Unformatted record is too short for input list
159	Trailing sign or "CR" not at end of field in business editing descriptor
160	Multiple leading sign before "\$" in business editing descriptor
161	"*" must precede "\$" or "Z" in business editing descriptor
162	"\$" in wrong position in business editing descriptor
163	"Z" after decimal point in business editing descriptor
164	Decimal point appears more than once in business editing descriptor
165	Comma at start of field or after decimal point in business editing descriptor
166	Invalid character found in business editing descriptor
167	DO-loop will never be executed (/DOCHECK)
168	Unanticipated DOS error encountered in I/O system
169	Underflow detected by input/output routine (data out-of-range)
170	Equals missing
171	Absolute value of complex argument out of range
172	The left hand side of a LET must be a variable or array element
173	You may not delete a file which is 'READONLY'
174	Array has wrong number of dimensions
175	Array subscript(s) out-of-bounds
176	Unpaired quotes
177	Name longer than 32 characters
178	Variable is not an array

179	Variable is an array
180	Unknown variable
181	Block IF unterminated on leaving a macro
182	Error in the structure of WHILE-ENDWHILE block in a macro
183	Error in the structure of block IF in a macro
184	Display full
185	Routine not found
186	Unknown vector
187	Parameters may not be altered
188	Too many points to be plotted
189	ANSI - invalid FORM specifier
190	Attempt to read from a file opened with FORM='PRINTER'
191	Key name expected

Error and exception handling (Win32)

Exceptions are events generated outside the normal flow of control through a program or thread of execution. Such an event may arise due to a hardware event (such as a page fault) or through a software trap such as an attempt to access another processes memory space. The default action of the process is to terminate the process and produce diagnostic information. Exceptions occur for the following events:

- ☐ Denormal floating point operand
- ☐ Floating point divide by zero
- ☐ Inexact floating point result
- ☐ Invalid floating point operation
- ☐ Floating point overflow
- ☐ Floating point stack overflow
- ☐ Floating point underflow
- ☐ Integer divide by zero
- ☐ Integer overflow
- ☐ Integer underflow
- ☐ Access violation
- ☐ Breakpoint
- ☐ Single step
- ☐ Execution of a privileged instruction

These exceptions can be split into three distinct groups: Floating point math exceptions, integer math exceptions and debugger exceptions.

FTN77 provides the programmer with a method to trap these exception events and to act appropriately. This means that it is possible to trap an underflow event and reset a variable to a known (say zero) value.

This is achieved by maintaining a table of functions to be executed in the event of an exception. Only one exception handler may be installed for any particular exception event at any one time. So you may have two different handlers installed for two different exception events, but you may not have two handlers chained together for the same exception event. This also applies to mixed language programming where nominally different handlers are required for Fortran and C code. If you want to handle an exception differently in different parts of the code, you can remove one exception handler and install another.

Each exception event is identified by an exception event code. This is an integer value that is used to uniquely identify each of the possible exceptions that are trapable by the user. These codes are defined in the insert file *except.ins* which is provided as part of the FTN77 system.

When an exception event occurs, the operating system copies the machine state into an area of memory. The image of the machine may be manipulated to correct the fault in order to resume execution in an orderly manner. Once the machine state has been saved, the exception handler searches for a handler offering the event to the following processes:

- ☐ Debugger first chance.
- ☐ The frame based handler installed by the program.
- ☐ Debugger second chance.

The frame based handler is the one installed by any main program compiled with FTN77. This handler is really a filter. It examines the exception event that has occurred and looks to see if the user program has installed a handler for that event. If such a handler routine is installed, control is passed back to the routine. If no handler is found, the Fortran program takes the default action or it terminates and the exception details are displayed for debugging purposes.

Here is a summary of the FTN77 error and exception handling routines that are peculiar to Win32. Details are given in the *FTN77 Library Reference*.

ACCESS_DETAILS@	To get details of the access violation.
CLEAR_FLT_UNDERFLOW@	To clear a floating point underflow exception.
EXCEPTION_ADDRESS@	To find the address of the instruction that generated the exception.
GET_VIRTUAL_COMMON_INFO@	To get virtual common block details.
PRERR@	To print the error message associated with a given error code.
RESTORE_DEFAULT_HANDLER@	To remove a user defined exception handler.
TRAP_EXCEPTION@	To install a user defined exception handler.

Overview of the FTN77 run-time library

This chapter contains outline information about the routines that are available in the FTN77 run-time library. Further information is available in *FTN77 Library Reference* and in the on-line Help system (in some cases a reference to MS-DOS should be replaced by the appropriate operating system). The routines below are arranged in functional groups with the given headings. Within the groups the routines are arranged in alphabetical order.

The following symbols are used to denote the availability of each function on the various platforms:

- no symbol Function is available on all platforms, DOS, Win16 and Win32.
- ❶ At the time of going to press, function is only available under DOS .
- ❷ Function is only available under DOS.
- ❸ Function is only meaningful under DOS. Under Win16 and Win32 the function either has no operation or is not relevant. This category is for DOS programs and programs that are being ported from DOS to Windows.
- ❹ Function is available under DOS and also in ClearWin+ but with slightly different functionality. See the *FTN77 Library Reference* for the DOS function and the Clearwin+ documentation (the manual or an information file on the release disk) for information on the ClearWin+ variant.
- ❺ Function is only available under Win32.

Index

	<i>page</i>
Character-handling	336
Data sorting.....	337
Error and exception handling	338
File manipulation	338
Graphics.....	340
Graphics plotter/screen.....	341
Graphics printer.....	342
Mouse	342
Printer	344
Process control.....	344
Random numbers	344
Sound	345
Storage management	346
System information.....	346
Text screen/keyboard	347
Text windows.....	348
Time and date	348

Bit-handling

CLEAR_BIT@	Clears the N'th bit of an array.
SET_BIT@	Sets the N'th bit of an array.
TEST_BIT@	Tests if the N'th bit of an array is set.

Character-handling

ALLOCSTR@	To allocate dynamic storage and copy a string.
APPEND_STRING@	Adds a string to the end of a line.
CENTRE@	Positions a string in the centre of a field.
CHAR_FILL@	To fill a string with a particular character.

CHSEEK@	Looks for a given string in an ordered array.	
CNUM	Converts an integer to character form.	
COMPRESS@	Compresses a string by using tabs.	
GETSTR@	To get a string that was stored using ALLOCSTR@	5
LCASE@	Alters a character argument so that all letters become lower case.	
NONBLK	Obtains the position of the first character that is not a space.	
SAYINT	Returns an integer argument as text.	
TRIM@	Removes leading spaces.	
TRIMR@	Rotates a character string right until there are no trailing spaces.	
UPCASE@	Alters a character argument so that all letters become upper case.	

Command line parsing

CMNAM	Reads a token from the command line.	
CMNAM@	Reads a token from the command line.	
CMNAMR	Resets the command line.	
CMNARGS@	To get the number of command line arguments.	5
CMNUM@	To get the next command line argument as an integer.	
CMPROGNM@	To get the program name.	5
COMMAND_LINE	Reads the whole command line.	
GET_PROGRAM_NAME@	Returns the name of the current program.	
SET_COMMAND_LINE@	To set the whole command line.	2

Data sorting

CHSORT@	Sorts an array of characters.
DSORT@	Sorts a REAL*8 array.
ISORT@	Sorts an integer array.
RSORT@	Sorts a REAL*4 array.

Error and exception handling

ACCESS_DETAILS@	To get details of the access violation.	5
CLEAR_FLT_UNDERFLOW@	To clear a floating point underflow exception.	5
DOS_ERROR_MESSAGE@	Gets a DOS error message.	
DOSERR@	Prints a DOS error message when an error occurs.	
ERR77	Prints a DOS error message and terminate a program when an error occurs.	
ERROR@	Prints a user defined error message and terminate a program.	
EXCEPTION_ADDRESS@	To find the address of the instruction that generated the exception.	5
FORTTRAN_ERROR_MESSAGE@	Gets a Fortran error message.	
GET_VIRTUAL_COMMON_INFO@	To get virtual common block details.	5
JUMP@	Executes a non-local jump.	
LABEL@	Sets a label for a non-local jump.	
PERMIT_UNDERFLOW@	Switches off floating point underflow checking.	
PRERR@	To print the error message associated with a given error code.	
QUIT_CLEANUP@	Prints a message and exit from a program with Control-break	
RESTORE_DEFAULT_HANDLER@	To remove a user defined exception handler.	5
RUNERR@	Prints the run-time error corresponding to a given IOSTAT value.	
SET_TRAP@	To trap a given event.	1
TRAP_EXCEPTION@	To install a user defined exception handler.	
UNDERFLOW_COUNT@	Gets the number of floating point underflows.	

File manipulation

ATTACH@	Sets the current directory.	
CLOSEF@	Closes a file.	
CLOSEFD@	Closes and delete a file.	
CLOSEV@	Closes a file opened with OPENV@	2
CURDIR@	Gets the current directory.	

CURRENT_DIR@	Obsolete routine. Use CURDIR@	5
DIRENT@	To obtain directory information.	
EMPTY@	Clears a file for writing.	
ERASE@	Deletes a file.	
FEXISTS@	To search for a file with a given path name or wildcard.	5
FILE_EXISTS@	Obsolete routine. Use FEXIST@ instead	5
FILE_SIZE@	To get the size of a file in bytes.	
FILE_TRUNCATE@	To truncate an open file at its current position.	
FILEINFO@	To get information about a specified file.	5
FILES@	Obtains directory information.	
FPOS@	Repositions a file.	
FPOS_EOF@	To move the file pointer to end-of-file.	5
GET_FILE_DATE_TIME_STAMP@	Gets the DOS date and time stamp for a particular file.	2
GET_FILES@	To get a list of files in the current working directory.	5
GET_PATH@	Gets the fully qualified pathname.	
GET_PATHV@	Gets the fully qualified pathname.	2
MKDIR@	Creates a new DOS directory.	
OPENR@	Opens a file for reading.	
OPENV@	To open a file for reading.	2
OPENRW@	Opens a file for reading or writing.	
OPENW@	Opens a file for writing.	
READF@	Reads binary data from a file.	
READFA@	Reads ASCII text from a file.	
RENAME@	Renames a file.	
RFPOS@	Gets the position of a file.	
SELECT_FILE@	To select from a displayed list of files.	2
SET_FILE_ATTRIBUTE@	Sets a file attribute.	
SET_SUFFIX@	Changes the extension of a given file name.	
SET_SUFFIX1@	Adds an extension to a given file name.	
TEMP_FILE@	Provides a unique name for a file.	
WILDCHECK@	To check for the matching of a file name with a wild card.	5
WRITEF@	Writes binary data to a file.	
WRITEFA@	Writes a line of data to an ASCII file.	

Graphics

CLEAR_SCREEN@	Clears the screen.	
CLEAR_SCREEN_AREA@	Clears a rectangular area of the screen.	
COMBINE_POLYGONS@	Gets the handle for a combination of polygons.	
CREATE_POLYGON@	Gets a handle for a specified polygon.	
DELETE_POLYGON_DEFINITION@	Deletes a polygon definition.	
DRAW_HERSHEY@	Draws an Hershey character.	
DRAW_LINE@	Draws a straight line in graphics mode.	
DRAW_TEXT@	Draws text in graphics mode.	
EGA@	Switches to EGA graphics mode.	④
ELLIPSE@	Draws an ellipse.	
FILL_ELLIPSE@	Fills an ellipse.	
FILL_POLYGON@	Fills a polygon.	
FILL_RECTANGLE@	Fills a rectangle.	
GET_ALL_PALETTE_REGS@	Gets all palette registers for colour graphics.	④
GET_DEVICE_PIXEL@	Gets the pixel colour for a virtual screen or printer.	④
GET_GRAPHICS_MODES@	Gets details of all the graphics modes.	③
GET_GRAPHICS_RESOLUTION@	Gets details of the high resolution graphics mode.	③
GET_PIXEL@	Gets a pixel colour.	④
GET_TEXT_MODES@	Gets information about the available text modes.	②
GET_TEXT_SCREEN_SIZE@	Gets the resolution of the current text mode.	②
GET_VIDEO_DAC_BLOCK@	Gets a block of VGA DAC registers.	④
GRAPHICS_MODE_SET@	Sets the graphics mode to a given resolution.	④
GRAPHICS_WRITE_MODE@	Selects replace/XOR mode before writing to the screen, virtual screen or printer.	
HERSHEY_PRESENT@	Tests if a character number has a Hershey representation.	
HIGH_RESOLUTION_GRAPHICS_MODE@	Switches to high resolution graphics mode.	④
IS_TEXT_MODE@	Tests if the screen is in text or graphics mode.	②
LOAD_STANDARD_COLOURS@	Loads the standard colours for 256 colour mode.	④
MOVE_POLYGON@	Moves the position of a polygon.	
POLYLINE@	Draws a number of connected straight lines.	
RECTANGLE@	Draws a rectangle.	
RESTORE_GRAPHICS_BANK@	Restores the graphics bank after a BIOS call.	③

RESTORE_TEXT_SCREEN@	Restores a text screen saved with SAVE_TEXT_SCREEN@.	2
SAVE_TEXT_SCREEN@	Saves the whole of the text screen.	2
SCREEN_TYPE@	Gets the graphics screen type.	2
SET_ALL_PALETTE_REGS@	Sets all palette registers for colour graphics.	4
SET_DEVICE_PIXEL@	Sets a pixel colour for a virtual screen or printer.	
SET_PALETTE@	Sets a palette register for colour graphics.	4
SET_PIXEL@	Sets a pixel to a colour.	4
SET_TEXT_ATTRIBUTE@	Sets the current graphics text attributes.	
SET_VIDEO_DAC@	Sets a VGA DAC register.	4
SET_VIDEO_DAC_BLOCK@	Sets a block of VGA DAC registers.	4
TEXT_MODE@	Returns to text mode.	3
TEXT_MODE_SET@	Selects the current text mode.	4
USE_VESA_INTERFACE@	Forces the VESA interface to be used.	3
VGA@	Switches to VGA graphics mode.	4

Graphics plotter/screen

CLOSE_PLOTTER@	Closes the plotter device or file.	4
CLOSE_VSCREEN@	Closes the virtual screen.	
CREATE_SCREEN_BLOCK@	Creates a screen block in memory.	4
GET_DACS_FROM_SCREEN_BLOCK@	Uses palette information from a PCX file.	
GET_SCREEN_BLOCK@	Saves a rectangular area of the screen.	
NEW_PAGE@	Provides a new page on the current graphics device.	
OPEN_PLOT_DEVICE@	Opens the plotter.	4
OPEN_PLOT_FILE@	Directs plotter output to a file.	4
OPEN_VSCREEN@	Opens a screen block as the virtual screen.	
PCX_TO_SCREEN_BLOCK@	Loads a file a screen block.	4
PLOTTER_SET_PEN_TYPE@	Selects a pen type for the plotter.	3
RESTORE_SCREEN_BLOCK@	Displays a previously saved area of the screen.	
SCREEN_BLOCK_TO_PCX@	Saves a screen block a file.	
SCREEN_BLOCK_TO_VSCREEN@	Loads a screen block to the virtual screen.	
SCREEN_TO_VSCREEN@	Loads the graphics screen to the virtual screen.	
VSCREEN_TO_PCX@	Saves the virtual screen to a file.	

VSCREEN_TO_SCREEN@
WRITE_TO_PLOTTER@

Loads the virtual screen to the graphics screen.
Writes a string to the plotter.

③

Graphics printer

CLOSE_GRAPHICS_PRINTER@
GET_PCL_PALETTE@

Closes the graphics printer device or file.
Gets the colour definitions for a given number of colours.

LOAD_PCL_COLOURS@
OPEN_GPRINT_DEVICE@
OPEN_GPRINT_FILE@

Loads the standard colour definitions.
Opens a graphics printer.
Directs graphics printer output to a file.

④

④

④

PRINT_GRAPHICS_PAGE@
SELECT_DOT_MATRIX@

Prints a graphics page.
Selects an Epson compatible dot matrix printer

②

SELECT_PCL_PRINTER@

Specifies attributes of a PCL printer.

④

SET_PCL_BITPLANES@

Sets the number of colours in the image.

③

SET_PCL_GAMMA_CORRECTION@

Alters the “gamma correction” for colours.

③

SET_PCL_GRAPHICS_DEPLETION@

Improves the image quality.

③

SET_PCL_GRAPHICS_SHINGLING@

Makes a number of print passes.

③

SET_PCL_LANDSCAPE@

Sets LANDSCAPE or PORTRAIT orientation.

③

SET_PCL_PALETTE@

Loads the colour definitions.

③

SET_PCL_RENDER@

Sets the “rendering algorithm”.

③

Hot key

DEFINE_HOT_KEY@
REMOVE_HOT_KEY@
FEED_KEYBOARD@

To associate a hotkey routine with a given key.

②

To disassociate a hotkey routine from a given key.

②

To push a keycode into the keyboard buffer.

②

In-line

FILL@
IN@

Sest an array of N bytes to a particular value.

②

To input one byte from an I/O port.

MATCH@	Compares two arrays of N bytes.	
MOVE@	Copies an array of N bytes.	
OUT@	To output one byte of data to an I/O port.	2
POP@	Pops a value off the system stack.	
PUSH@	Pushes a value on the system stack.	
SET_IO_PERMISSION@	To set the I/O permission level to 3 or 0.	2

Mouse

DISPLAY_MOUSE_CURSOR@	Shows the mouse cursor on the screen.	3
GET_MOUSE_BUTTON_PRESS_COUNT@	Gets the number of times a button has been pressed.	
GET_MOUSE_EVENT_MASK@	Gets the mask for the most recent mouse interrupt.	
GET_MOUSE_PHYSICAL_MOVEMENT@	Gets the mouse pad distance from the last call.	3
GET_MOUSE_POSITION@	Gets the present state of the mouse cursor.	
GET_MOUSE_SENSITIVITY@	Gets the values of the physical movement ratios and the double speed threshold.	3
HIDE_MOUSE_CURSOR@	Hides the mouse cursor on the screen.	3
INITIALISE_MOUSE@	Initialises the mouse driver.	3
MOUSE@	Performs a mouse interrupt.	3
MOUSE_CONDITIONAL_OFF@	Switches off the cursor when it enters a specified rectangle.	3
MOUSE_LIGHT_PEN_EMULATION@	Uses the mouse as a light-pen.	3
MOUSE_SOFT_RESET@	Initialises the mouse software.	
QUERY_MOUSE_SAVE_SIZE@	Gets the buffer size for the mouse state.	3
RESTORE_MOUSE_DRIVER_STATE@	Restores a former state of the mouse driver.	3
SAVE_MOUSE_DRIVER_STATE@	Saves the current state of the mouse driver.	3
SET_MOUSE_BOUNDS@	Restricts mouse movements to a specified rectangle.	3
SET_MOUSE_GRAPHICS_CURSOR@	Specifies the shape of the mouse cursor for graphics mode.	3
SET_MOUSE_INTERRUPT_MASK@	Enables mouse actions to produce interrupts.	
SET_MOUSE_MOVEMENT_RATIO@	Sets the mouse cursor sensitivity.	3
SET_MOUSE_POSITION@	Moves the mouse cursor to a particular position.	

SET_MOUSE_SENSITIVITY@	Sets the mouse cursor sensitivity and the threshold for the double speed.	③
SET_MOUSE_SPEED_THRESHOLD@	Sets the threshold for double speed.	③
SET_MOUSE_TEXT_CURSOR@	Specifies details of the mouse cursor for text mode.	③

Printer

PRINT_CHARACTER@	To send one character to the printer.	②
INITIALISE_PRINTER@	To initialise the printer.	②
GET_PRINTER_STATUS@	To obtain status information for the printer.	②

Process control

CISSUE	Issues a DOS command.	
EXIT	Terminates a program.	
EXIT@	Terminates a program.	
SLEEP@	Suspends program execution for a specified time interval.	
SPAWN@	Initiates a concurrent subtask.	②
START_PROGRAM@	Starts another Salford program.	②
YIELD@	To yield control to a subtask.	②

Random numbers

DATE_TIME_SEED@	Selects a new “seed” for the pseudo-random number generator function RANDOM.
RANDOM	Returns a pseudo-random double precision value.
SET_SEED@	Enters a new “seed” for the pseudo-random number generator function RANDOM.

Real mode

ALLOCATE_REAL_MODE_MEMORY@	To allocate real mode memory.	②
COPY_FROM_REAL_MODE@	To copy data from a real mode program.	②
COPY_FROM_REAL_MODE1@	To copy data from a real mode program.	②
COPY_FROM_SEGMENT@	To copy data from another segment.	②
COPY_TO_REAL_MODE@	To copy data to a real mode program.	②
COPY_TO_REAL_MODE1@	To copy data to a real mode program.	②
COPY_TO_SEGMENT@	To copy data to another segment.	②
DEALLOCATE_REAL_MODE_MEMORY@	To free real mode memory.	②
DOSCOM@	To obtain a segment selector for the DOSCOM buffer.	②
FTN77WT etc.	Used within a real mode program to receive control from and return control to a FTN77 program.	②
LINEAR_ONE_MEG_SEG@	To obtain the real mode address 0.	②
LOAD_REAL_MODE_LIBRARY@	To load and execute a real mode program.	②
MODIFY_REAL_MODE_MEMORY@	To change the size of a block of real mode memory.	②
REAL_MODE@	To transfer control from a FTN77 to a real mode program.	②
REAL_MODE_ADDRESS_OF_DOSCOM@	To obtain the address of the DOSCOM buffer.	②
REAL_MODE_INTERRUPT@	To cause a real mode interrupt from an FTN77 program.	②
SCREENSEG@	To obtain the segment selector for the graphics area.	②

Sound

BEEP@	Outputs an audible beep.	②
SOUND@	Makes an audible sound at the console.	②

Storage management

FREE_SPACE_AVAILABLE@	Obtains the amount of free memory in the system.	②
FREE_VIRTUAL_PAGES@	Frees memory for reuse.	②
GET_MEMORY_INFO@	Obtains information about the memory.	②
GET_STORAGE@	Gets a block of storage of size N bytes from the storage heap.	
GET_STORAGE1@	Gets a block of storage from the storage heap.	②
LARGEST_BLOCK_AVAILABLE@	Obtains the size of the largest free block in the storage heap.	②
MEMORY_AVAILABLE@	Gets the total size of available heap space.	②
RETURN_STORAGE@	Returns a block of storage.	
SET_PAGES_RESERVE@	Warns of a limited page reserve.	②
SET_TRAP_ON_PAGE_TURN@	Warns of the first page turn.	②
SHRINK_STORAGE@	Shrinks a block of storage.	
USE_STORAGE@	Offers additional memory to the storage heap.	②
USE_VIRTUAL_SCRATCH_FILES@	Enables or disable the virtual scratch file facility.	①

System information

DBOS_VERSION@	To get the current DBOS version number.	②
DOSPARAM@	To get an environment variable.	
DYNT@	To test for the presence of a system routine.	②
DYNT1@	To test for the presence of a user routine.	②
GET_COPROCESSOR_ENVIRONMENT@	To obtain the types of processors available on the system.	②
GET_CURRENT_FORTRAN_IO@	To access the state of the current Fortran I/O unit.	
GET_CURRENT_FORTRAN_UNIT@	To get the unit number for the current I/O operation.	
GETENV@	To get an environment variable.	⑤

Text screen/keyboard

COU@	Outputs text to the screen with a new line.	
COUA@	Outputs text to the screen without a new line.	
COUP@	Outputs text to a given screen position.	2
DOS_KEY_WAITING@	Tests if the keyboard buffer is empty.	2
ECHO_INPUT@	Controls the echoing of text from standard input.	2
ERRCOU@	Outputs text to the standard error device.	5
ERRCOUA@	Outputs text to the standard error device.	5
ERRNEWLINE@	Writes an newline to the standard error device.	5
ERRSOU@	Outputs text to the standard error device.	5
ERRSOUA@	Outputs text to the standard error device.	5
GET_CURSOR_POS@	Gets the co-ordinates of the text cursor.	2
GET_DOS_KEY@	Gets the next keycode.	2
GET_DOS_KEY1@	Gets the waiting keycode.	2
GET_EXTENDED_CHAR@	Gets the waiting two-byte keycode.	2
GET_KEY@	Gets the next keycode.	
GET_KEY1@	Gets the waiting keycode.	2
GETCL@	Gets a line of text from the keyboard.	2
HIDE_CURSOR@	Hides the text cursor.	2
KEY_WAITING@	Tests if the keyboard buffer is empty.	2
NEWLINE@	Writes a carriage return/linefeed to the screen (standard output).	
PRINT_BYTES@	Writes a sequence of hexadecimal values.	
PRINT_BYTES_R@	To write a hexadecimal sequence in reverse order.	
PRINT_HEX1@	Prints a 1 byte hexadecimal number (2 digits) without a new line.	
PRINT_HEX2@	Prints a 2 byte hexadecimal number (4 digits) without a new line.	
PRINT_HEX4@	Prints a 4 byte hexadecimal number (8 digits) without a new line.	
PRINT_I1@	Prints an INTEGER*1 decimal number without a new line.	
PRINT_I2@	Prints an INTEGER*2 decimal number without a new line.	
PRINT_I4@	Prints an INTEGER*4 decimal number without a new line.	
PRINT_R4@	Prints an REAL*4 decimal number without a new line.	
PRINT_R8@	Prints an REAL*8 decimal number without a new line.	
READ_EDITED_LINE@	Inputs text from a screen position.	2
RESTORE_CURSOR@	Shows the text cursor.	2
SET_CURSOR_POS@	Sets the co-ordinates of the text cursor.	2
SET_CURSOR_TYPE@	Sets the shape of the text cursor.	2

SOU@	Outputs text with a new line, omitting any trailing blanks.
SOUA@	Outputs text without a new line, omitting any trailing blanks.

Text windows

CONCEALW@	Moves a window to the bottom of the stack.	2
KILLW@	Removes a text window.	2
MOVEW@	Changes the position of a window on the screen.	2
POPW@	Moves a window to the top of the stack.	2
SCROLL_DOWN@ and SCROLL_UP@	Scrolls text in a window.	2
SET_CURSOR_POSW@	Sets the cursor position for a text window.	2
WBORDER@	Sets the border style for a text window.	2
WCLEAR@	Clears a text window	2
WCOU@	Writes text to a window.	2
WCOUP@	Writes text to a window position.	2
WCREATE@	Creates a text window.	2
WDBORDER@	Sets the default border style for all subsequent text windows created.	2
WDSHADOW@	Sets the default shadow style for all subsequent text windows created..	2
WMEMORY@	Gets the memory pointer for a text window.	2
WREAD_EDITED_LINE@	Inputs text from a window position.	2
WSHADOW@	Sets the shadow style for a text window.	2
WTITLE@	Assigns a title to a text window.	2

Time and date

CLOCK@	Gets a time in seconds.	
CONVDATE@	To get the date in numeric form.	5
DATE@	Gets the date in the form MM/DD/YY (American format).	
DCLOCK@	Gets a time in seconds.	
EDATE@	Gets the date in the form DD/MM/YY (European format).	

FDATE@	Gets the date in text form.	
HIGH_RES_CLOCK@	To obtain the CPU time accurate to 1 microsecond.	
SECONDS_SINCE_1980@	Gets the number of seconds from a fixed date.	
SET_ALARM_CLOCK@	To set the elapsed time before an alarm.	2
TIME@	Gets the time in the format HH:MM:SS.	
TODATE@	To convert a given time to a date in the form MM/DD/YY.	5
TOEDATE@	To convert a given time to a date in the form DD/MM/YY.	5
TOFDATE@	To return the date in text form.	5
TOTIME@	To return the time in the form HH:MM:SS.	5

Index

*,LINK77 command, 232

—

__stdcall symbols, 262

_SALFStartup entry point for executables, 262

A

ANSI conformity, 28

ANSI directive, 177

ANSI,compiler option, 28, 30, 31, 43, 132, 140, 175, 177

APPEND_BINARY,compiler option, 43

APPEND_LIST,compiler option, 24, 43

Argument consistency,checking at run-time, 80

Argument,dummy array, 92

Arguments,character, 173

Arithmetic overflow,checking at run-time, 79

Array subscript checking at run-time, 81

Array used as actual argument, 80

Assembler 32-bit Intel, 24

Assembler comments, 194

Assembler labels, 195

Assigned GOTO checks at run-time, 83

Automatic loading and execution of programs, 39

B

B edit descriptor, 133, 183

Binary data values, 182

BINARY,compiler option, 29, 44

BIOS routines - how to call them, 304

BREAK,compiler option, 44, 304

BREAK,RUN77 option, 234

BRIEF,compiler option, 44

Business editing, 133

 ASTERISK (*), 134

 comma, 135

 CREDIT (CR), 135

 decimal point (.), 135

 DOLLAR sign (\$), 134

 MINUS sign (-), 134

 number sign (#), 135

 PLUS sign (+), 134

 ZED (Z), 134

C

C,compiler option, 47

C_EXTERNAL statement, 213

CCORE1 routine, 199

CELSE statement, 190

CENDIF statement, 190

Character

 arrays, 162

 assignments, 160

 comparisons, 167, 171

 constants, 159

 expressions, 159

 functions, 172

 input/output, 164

 substrings, 162

Character arguments,length of, 83

Character data,length of, 170

Character handling facilities, 157

Character variable,overheads when using long, 92

CHECK,compiler option, 30, 44, 79, 81, 181, 189, 205

Checking character data handling at run-time, 83

Checking substring expressions at run-time, 84

CIF statement, 190

CLOSE statement

 Description of specifiers, 116

 General form of, 116

 Status of files at program termination, 116

CMNAM@ routine, 42

CODE compiler directive, 194

Code motion, 88

COFF, 241

Comment message, 27

Common blocks in dynamic link libraries, 239

COMMON statement,character data, 175

COMMON_BASE,LINK77 command, 233

Compilation, 21

 conditional, 190

 listing, 23

 messages, 75

 suppressing the listing of, 35

Compiler directive, 34

 INCLUDE, 36

 INTL, 178

 INTS, 178

 LIST, 35

 LOGL, 178

 LOGS, 178

- NOLIST, 35
- OPTIONS, 35
- Compiler options
 - default, 49
 - reading from file, 33
- COMPLEX*16 data, 139
- CONFIG, compiler option, 32, 44, 49
- CONFIGDB command, 297
- Configuring DBOS, 297, 298
- Constant folding, 86
- CONTROL BREAK handler example, 206
- Coprocessor emulation, 5
- Coprocessor, use of, 5, 197
- CORE intrinsic functions, 199
- CORE1 routine, 199
- CORE2 routine, 199
- CORE4 routine, 199

D

- Data initialisation in type statement, 179
- DATA statement, 163
 - setting address constants with, 200
 - special form of, 189
- Data transfer statement
 - Description of specifiers, 122
 - Effect of first WRITE statement, 127
 - General form of, 122
- DBOS options
 - DISK_CACHE, 294
 - EXTMEM, 293
 - memory limits, 295
 - NO_SHIFT_INTERRUPTS, 294
 - NOWEITEK, 294
 - PAGE, 294
 - PRIMELINK, 294
 - SEARCHMEM, 293
 - USE_XMS, 295
- DBOS system, 193, 302
- DBOS_SET and DBOS_RESET commands, 298
- DBREAK, compiler option, 44, 304
- DCLVAR, compiler option, 28, 44
- DCORE8 routine, 199
- DEBUG, compiler option, 30, 44
- Debugging system
 - /BREAK option, 52
 - /DBREAK option, 52
 - invoking, 52
- DEFCOM, LINK77 command, 233
- Determination of storage address, 139
- Diagnostic facilities, 75
- Diagnostics, 2
 - compilation, 75
 - run-time, 79
- Direct access, 101, 129
- DISK_CACHE, DBOS option, 294
- DO WHILE statement, 184

- DO1, compiler option, 45
- DOCHECK, compiler option, 45
- DOS routines - how to call them, 304
- DOSCOM@ routine, 305
- DOUBLE PRECISION, automatic use of, 31
- DO-variable used as actual argument, 80
- DREAL, compiler option, 31, 45, 98
- Dynamic link libraries, 237
- Dynamic storage, 29

E

- EDOC compiler directive, 194
- Efficient use of Fortran 77, 85
- ENCODE Fortran 66 syntax, 180
- END DO statement, 185
- End-of-file condition, 103
 - to set from the screen, 106
- ENTRY, LINK77 command, 232
- EQUIVALENCE statement, error messages for, 76
- Error message, 27
- ERROR_NUMBERS, compiler option, 28, 45
- Execution errors, list of, 325
- EXIT statement, 186
- Expanded source listing, 1
- EXPLIST, compiler option, 24, 45, 206
- Extensions to the ANSI standard
 - List-directed input, 133
 - OPEN statement, 133
 - RECL specifier, 133
 - RENAME specifier, 133
- EXTERNAL statement, 139
- EXTMEM, DBOS option, 293
- EXTREFS, compiler option, 45

F

- FCORE4 routine, 199
- File existence, 99
- File names, 99
- File Positioning statements
 - BACKSPACE statement, 132
 - Description of specifiers, 131
 - ENDFILE statement, 132
 - General form of, 131
 - REWIND statement, 132
- File properties, 99
- File structure, 99
- FILE, LINK77 command, 230
- Filename in the OPEN statement, 107
- FORCE_LOAD, LINK77 command, 230
- FORMAT statement, efficient use of, 93
- Formats, contained in non-character arrays, 137
- Fortran 77 extensions, 177
 - FORMAT statement, 137
 - Input/output, 132
- Fortran compilers other than FTN77, 30
- FTN77

- peep-hole optimisations in, 85
- simple use of, 16
- treatment of common subexpressions in, 92
- treatment of constants in, 92

FTN77, simple use of, 8

FULLCHECK, compiler option, 45, 79, 81, 181, 189, 205

FULLDCLVAR, compiler option, 45

FULLMAP, compiler option, 45

FULLXREF, compiler option, 45

G

General Protection Exceptions, 303

GUI, 223

H

HARDFAIL, compiler option, use with load-and-go, 41

HARDFAIL, compiler option, 45

HARDFAIL, RUN77 option, 234

HELP, compiler option, 13, 45

HELP77 utility, 13

Hexadecimal data values, 182

Hollerith data, 180

I

Identifier, internal files, 103

IGNORE, compiler option, 28, 45, 77

IMPLICIT NONE, compiler directive, 191

IMPLICIT_NONE, compiler option, 45

INCLUDE, compiler directive, 36

INCLUDE, LINK77 command, 231

INDEX, usage of intrinsic function, 170

Induction weakening, 88

Initial point, file position, 101

In-line assembler

- literals in, 196

Input/output specifier

- ACCESS=, 108

- BLANK=, 110

- DRIVER=, 109

- END=, 105

- ERR=, 104

- FILE=, 107

- FILETYPE=, 108

- FORM=, 109

- IOSTAT=, 104

- RECL=, 110

- RENAME=, 117

- STATUS=, 107, 116

- UNIT=, 103

Input/output statements, 95

- ACCESS=, 120

- BLANK=, 121

- DIRECT=, 120

- End of record terminator, 98

END=, 127

ERR=, 119, 126

EXIST=, 119

FMT=, 123

FORM=, 120

FORMATTED=, 120

FUNIT=, 121

IOSTAT=, 119, 126

NAME=, 119

NAMED=, 119

NEXTREC=, 121

NML=, 124

NUMBER=, 119

OPENED=, 119

Permitted specifiers with, 96

REC=, 127

RECL=, 120

SEQUENTIAL=, 120

UNFORMATTED=, 120

INQUIRE statement

- Additional FTN77 feature of, 121

- Description of specifiers, 119

- examples in the use of, 121

Integer data, long and short, 178

INTEGER* statement, 178

Internal files, 101

INTERNAL PROCEDURE

- examples in the use of, 188

Interrupt routines, 191

INTL, compiler option, 30, 46, 98, 141, 178

Intrinsic function, 139, 169

- character manipulation with, 169

- FTN77-specific, 185

- generic name for, 140

- inline code for, 91

- integer arguments and results, 141

- logical arguments and results, 142

- names not allowed as actual argument, 141

- names used as actual argument, 141

- non-ANSI, 139

- notes, 148

- specific name for, 140

INTRINSIC statement, 139

INTS, compiler option, 30, 98, 178

IOSTAT values, list of, 325

K

KILL_DBOS command, 7

L

Language extensions

- input/output, 183

LARGE_FILE, LINK77 command, 231

Length of variable names, 182

LGO,compiler option, 8, 39, 46, 303, 304
LIBOFFSET,LINK77 command, 238
Libraries
 dynamic link, 237
 relocatable binary, 235
LIBRARY,compiler directive, 41, 78
LIBRARY,compiler option, 41, 46
LINK77 commands, 230
 *, 232
 COMMON_BASE, 233
 DEFCOM, 233
 ENTRY, 232
 FILE, 230
 FORCE_LOAD, 230
 INCLUDE, 231
 LARGE_FILE, 231
 LIBOFFSET, 238
 LOAD, 230
 LOAD_EXHAUSTIVE, 230
 MAP, 231
 NOSUPPRESS, 232
 NOTIFY, 231
 PERMIT_DUPLICATES, 231
 PRESERVE_CASE, 232
 QUIT, 231
 REPORT_DEBUG_FILES, 232
 SUPPRESS, 232
 SUPPRESS_COMMON_WARNINGS, 232
 SYMBOL, 233
 XREF, 231
LINK77,compiler option, 40, 46, 229
LIST,compiler directive, 35
LIST,compiler option, 23, 46
List-direct I/O, use with internal files, 137
LOAD,LINK77 command, 230
LOAD_EXHAUSTIVE,LINK77 command, 230
Load-and-go facility, 39
Loader diagnostics, 78
Loading, 21
Loading FORTRAN programs using LINK77, 230
LOC intrinsic function, 200
Logical operations,bitwise, 139
LOGICAL* statement, 178
LOGL,compiler option, 31, 46, 98, 178
LOGS,compiler option, 31, 98, 178
Long variable names, 182
Loop invariants, 88

M

MAKE utility, 265
MAKEDA77 command, 99, 130
MAP,compiler option, 25, 46, 204
MAP,LINK77 command, 231
memory limits,DBOS option, 295
Memory map for DBOS, 306
MKLIB,compiler option, 46

MKLIB77
 command mode, 236
 interactive mode, 236
Multiple opening of a file, 102

N

Namelist-directed I/O, 124
Negative addresses, 302
New line, suppression in FORMAT statement, 137
NO_BINARY,compiler option, 29, 46
NO_COMMENTS,compiler option, 46
NO_CR,compiler option, 29, 46
NO_FAIL,compiler option, 46
NO_FLOATING_TRACKING,compiler option, 46, 88, 94
NO_OFFSET,compiler option, 24, 46
NO_OPTIMISE,compiler option, 86
NO_PEEP_HOLE,compiler option, 47
NO_SHIFT_INTERRUPTS,DBOS option, 294
NO_WARN73,compiler option, 28, 47
NO_WARNINGS,compiler option, 47
NO_WEITEK,compiler option, 86
NOLINK,compiler option, 47
NOLIST,compiler directive, 35
NOSUPPRESS,LINK77 command, 232
NOTIFY,LINK77 command, 231
NOTRACKING,compiler option, 47
NOWEITEK,DBOS option, 294
Numeric checking of variables and array elements, 188
Numeric data,limits for, 79

O

O edit descriptor, 183
Object code
 properties of, 29
Octal data values, 182
OLDARRAYS,compiler option, 47, 82
OMF, 241
ONLY_UNDEF,compiler option, 47
OPEN statement
 Additional FTN77 features of, 108, 110
 Description of specifiers, 107
 DRIVER keyword, 109, 113
 examples in the use of, 111
 general form of, 106
 specification of device drivers with, 109, 113
Optimisation, 85
OPTIMISE,compiler option, 47, 85, 86, 202
OPTIONS,compiler directive, 35, 79
OPTIONS,compiler option, 33, 47

P

Page memory exhausted, 301
PAGE,DBOS option, 294, 299
PAGETHROW,compiler option, 23, 47
Paging algorithm, 299

PARAMETER names in FORMAT statements, 137
 PARAMETER statement, 161
 PARAMS compiler option, use with load-and-go, 42
 PARAMS, compiler option, 47
 PARAMS, RUN77 option, 234
 PE, 241
 PERMIT_DUPLICATES, LINK77 command, 231
 PERSIST, compiler option, 24, 47, 75
 Plotter Interfacing, 313
 Portable Executable, 241
 PRELOAD, RUN77 option, 234
 PRESERVE_CASE, LINK77 command, 232
 PRIMELINK, DBOS option, 294
 Printer, writing directly to, 109
 PROFILE, compiler directives, 37
 PROFILE, compiler option, 37, 48
 Program development, 75
 Protected mode, 193, 302

Q

QUIT, LINK77 command, 231

R

Range-check for numeric variables, 189
 READ, RUN77 option, 235
 READONLY status, 102, 108
 READU, RUN77 option, 235
 Real mode, 193, 302
 Real mode program calling, 315
 REAL* statement, 179
 Record types
 Endfile, 98
 Formatted, 98
 Unformatted, 96
 Register
 dumps, 304
 locking, 89
 tracking, 87
 Relocatable binary, 29
 REPORT_DEBUG_FILES, LINK77 command, 232
 Resource compiler, 19
 Routines
 Character-handling, 336
 Data sorting, 337
 File manipulation, 338
 Graphics, 340
 Graphics plotter/screen, 341
 Graphics printer, 342
 Mouse, 343
 Printer, 344
 Process control, 344
 Random numbers, 344
 Sound, 345
 Storage management, 346
 System information, 346

Text screen/keyboard, 347
 Text Windows, 348
 Time and date, 348
 RUN77 options
 BREAK, 234
 HARDFAIL, 234
 PARAMS, 234
 PRELOAD, 234
 READ, 235
 READU, 235
 WRITE, 235
 WRITEU, 235
 RUN77 utility, 234
 RUNERR@ routine, 105

S

SALFLIBC.DLL, 249
 SALFLIBC.LIB, 249
 SAVE, compiler option, 29, 48
 SCREENSEG@ routine, 306
 SEARCHMEM, DBOS option, 293
 Sequential access, 101, 128
 Sequential unformatted record structure, 100
 SET_TRAP@ routine, 191, 200, 206, 301
 Shifts, bitwise, 139
 SILENT, compiler option, 27, 48
 SLINK
 Abbreviating commands, 243
 Archives, 249
 Command line mode, 242
 Comment text, 247
 Comments, 244
 data, 253
 Differences between command line mode and interactive mode, 244
 Direct linking with DLLs, 246
 Dynamic link libraries, 250
 Entry Points, 262
 entryname, 252
 Generation of archives, 250
 Generation of DLLs and exporting of functions, 251
 Import Libraries, 249
 internalname, 252
 Link map, 245
 Linking for Debug, 247
 Linking multiple object files, 243
 Mixing command line script files and interactive mode script files, 244
 Runtime tracebacks, 246
 Script or command files, 243
 Standard libraries and import libraries, 249
 The export command, 252
 Unresolved externals, 245
 Virtual Common, 248

SLINK command

- addobj, 250, 253
- archive, 253
- comment, 247, 253
- debug, 247
- decorate, 253
- dll, 254
- entry, 254
- export, 251
- exportall, 251, 255
- exportx, 251, 255
- file, 255
- filealign, 256
- heap, 256
- imagealign, 257
- imagebase, 257
- load, 258
- lure, 245, 257
- map, 258
- notrace, 247, 258
- quit, 259
- stack, 258
- subsystem, 259
- virtualcommon, 248, 258

Source file, 22

SPARAM, compiler option, 48, 190

SPECIAL ENTRY statement, 200

SPECIAL PARAMETER statement, 190

SPECIAL SUBROUTINE statement, 200

Compilation, 1

SRC, 19

STACK, compiler option, 32, 48

Statement execution count, 36

Statement function, inline code for, 92

Statement label, use of, 91

Static storage, 29

Statistics, compilation, 27

STATISTICS, compiler option, 28, 48

STDCALL statement, 223

Subroutines

- arguments, 202

Substring, 162

SUPPRESS, LINK77 command, 232

SUPPRESS_COMMON_WARNINGS, LINK77 command, 232

SVC pseudo-op's, 303

SVC/3, 305

SYMBOL, LINK77 command, 233

T

Terminal point, file position, 101

TOUCH utility, 267, 269, 270

TRAP routines in DBOS, 206

U

UNDEF, compiler option, 30, 48, 79, 82, 300

Undefined variables, checking for, 82

UNDERFLOW compiler option, use with load-and-go, 42

UNDERFLOW, compiler option, 48

Unit numbers, permitted values for, 102

UNSAFE, compiler option, 89

Use of the characters @ \$ and _, 182

USE_XMS, DBOS option, 295

V

Variables

- common, 202
- dynamic, 201
- static, 202

Virtual address space, 302

W

Warning message, 27

Weitek coprocessor, 91, 197, 201

WEITEK, compiler option, 32, 48, 86

WHILE statement, 184

Windows 3.x, running DBOS applications with, 307

WRITE, RUN77 option, 235

WRITEU compiler option, use with load-and-go, 42

WRITEU, RUN77 option, 235

X

XREF, compiler option, 25, 49

XREF, LINK77 command, 231

Z

Z edit descriptor, 183

ZEROISE, compiler option, 30, 49, 80