FTN95™

User's Guide

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 9, 2000

Preface

This user's guide describes the facilities available in version 2.04 and later of FTN95(Win32), the Salford Fortran 95 compiler for 80486 and Pentium based Personal Computers. This edition of the compiler is suitable for the Windows NT Operating System and for Windows 95/98. It can also be used with ClearWin+ in order to generate Win32 applications for Windows NT and Windows 95/98.

This guide also describes many of the facilities available in FTN95(DOS/Win16), the Salford Software Fortran 95 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 95/98. When used with Salford's ClearWin+, it can also be used to create Win16 applications for Windows 3.1(1) or Windows 95/98. Features of the DOS/Win16 edition that are peculiar to these operating systems are desribed in the *FTN95 User's Supplement* that can be found on the *Salford Tools CD*.

The guide concentrates on compiler-specific features and those areas of the Fortran language where the Fortran 95 Standard[1] needs amplification. It is not intended to be used as a Fortran language reference manual although Chapter 8 does contain a detailed guide to the features of input/output whilst Chapter 23 gives details of standard Fortran intrinsic functions.

FTN95 provides a large number of useful subroutines and functions in addition to those specified in the ISO Standard. These are outlined in Chapter 24 and described in the on-line Help system and in Chapter 25.

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

[1] ISO/IEC 1539-1:1997

Chapter headings in this guide:

Some chapters relate only to the Win32 edition of the compiler. These are distinguished in the even page header.

Acknowledgements

* * *

Table of Contents

1.

Introduction

Fortran 95 compiler

FTN95™ is a full Fortran 95 compliant compiler for Extended DOS, Windows 3.xx and Win32 (NT and 95/98). The compiler is delivered as a bundle comprising a Win32 edition and an Extended DOS/Windows 3.xx edition. FTN95 compilers are supplied with a fully-featured IDE, debugger, comprehensive compiler library (which includes graphics, operating system access, low-level file management, sorting, etc.), built-in 32 bit assembler, linker and Salford ClearWin+ (Salford's Windows GUI development library and tools).

Salford FTN95 features

- Very fast compilation speed (300,000 lines per minute have been measured compiling Fortran 77 on a Pentium Pro 200).
- Low-end PC system usability. The minimum PC is a 386 with 8MB RAM for the Extended DOS version.
- Full Fortran 95 source language and compatibility with all commonly available Fortran 77 source extensions, in particular, those available in FTN77.
- Mixed Fortran 77 and Fortran 95 programs (no need to recompile Fortran 77 code compiled with FTN77).
- Excellent compile-time diagnostics.
- Error-pinpointing run-time diagnostics (using one of the checking switches).
- Competitively fast run-time code (using the optimise switch).

New Fortran 95 language features

These include:

- FORALL statement and structure.
- ELEMENTAL/PURE subprograms.
- TYPE initialisers.
- The intrinsic NULL().
- Some array intrinisics have been extended, e.g. MAXLOC.

Special features of FTN95

1. Full support for REAL*10 and COMPLEX*20.
2. Compatibility with Salford FTN77 includes:
   - Binary compatibility with FTN77 object code even in UNDEF mode. Existing FTN77 libraries will work with FTN95 without recompilation.
   - Complete I/O library compatibility (FTN95 and FTN77 produced routines can share all I/O operations) which means previously created unformatted files may be used without recreation.
   - Inline Mnemonic Assembler using CODE ... EDOC.
   - C_EXTERNAL statement (even in MODULEs.). Full C/C++ interoperability (with Salford C++ and also Microsoft C++ under Win32).
   - ClearWin+ is Salford's GUI builder for Windows 3.1, Windows 95 and Windows NT. It is almost certainly the world's most powerful method for a Fortran user to program Windows. ClearWin+ users specify a GUI using formats linked typically to call-back functions which use the program's data from COMMON blocks or MODULEs.
   - OPTIONS statement (for compile-time switches etc.) and WINAPP statement (for Windows programs).
   - Optimised intrinsic library (many are inlined).
   - Optimised array features.
   - CHECK/UNDEF compile-time switches for automated run-time checking.
   - Full screen debuggers for DOS and GUI debugger for Windows.

- Small EXE and OBJ files.
- Supports all ‘deleted’ Fortran 95 features (e.g. REAL DO-loop indices).
- Most command-line compile-time switches are the same as those used by FTN77.

3.

Using /LGO and /LINK

Load and go

FTN95 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. /LINK has no effect when used with /LGO.

Most of the other compiler options summarised in chapter 4 are available 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:

```
FTN95 MYPROG /CHECK /LGO
```

would compile, load and execute the program held in the source file MYPROG.F90. 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

```
FTN95 MYPROG /LINK
```

the linker is automatically invoked after compilation is complete (assuming, of course, that no compilation errors have occurred). The resultant object code is loaded and a corresponding .EXE file is produced (a .OBJ file is not saved). 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 and page 13).

Relocatable binary libraries and input files

The use of the /LGO and /LINK options is not restricted to programs that require only the FTN95 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 FTN95 command line. For example:

```
FTN95 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 FTN95 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 FTN95 command does not contain the suffix.
- Dynamic link libraries are not specified on the LIBRARY directive. Under DOS/Win16 they are specified in the LIBRARIES.DIR file, see *FTN95 User's Supplement* for further details. Under Win32, DLLs are normally located either in the compiler directory or on the PATH.

The /HARDFAIL option

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 (within CODE/EDOC).

The /UNDERFLOW option

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 5). If underflows occur during program execution and the /UNDERFLOW option is not used, a message is output 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 FTN95 from scanning the whole command line before the user's program is executed.

For example, suppose that NEWPROG.F90 obtains two filenames FILE1 and FILE2 by means of calls to the system routine CMNAM. These filenames could be specified as follows:

```
FTN95 NEWPROG /LGO /PARAMS FILE1 FILE2
```

An illustrative program appears with the description of the CMNAM routine (see page 306).

4.

Compiler options

Quick reference

Compiler options are specified as part of the FTN95 command line, for example:

```
FTN95 MYPROG /ISO /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 FTN95 option /LGO, see chapter 3 for details. The interactive source level debugging system (see chapter 5) 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 chapter can be adopted as compiler defaults. The default options can be listed and changed by using the /CONFIG compiler option.

/132
: Prints error messages formatted for 132 column screens.

/ANSI
: Old synonym for /ISO option.

/ASMBREAK
: Like /BREAK but enters debugger at assembler level.

/BINARY <filename>
: Used to specify the .OBJ file name. If this option is omitted, the OBJ file name is created by replacing the source file-name's suffix with .OBJ, e.g. with FOO.F90, the object file would be FOO.OBJ.

/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 modified form that includes the full pathname of the file.

/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).

/CHECKMATE
Synonym for /FULL_UNDEF.

/CKC
Appends "__C" to a C_EXTERNAL alias which does not start with a double underscore.

/COLOUR
causes the compiler to output colour coded messages when it is called from a console. The default colours can be changed using the environment variable SALFORD_COLOUR (see page 357).

/CONFIGURE
Set-up installation compiler defaults. This must be the only option on the command-line.

/CONVERT
Converts fixed-format source into free-format, output to the screen. All comments are lost and Hollerith constants replaced with strings. This is useful for converting fixed-format INCLUDE files into files suitable for both fixed and free formats. This option implies the /NO_BINARY option and no error checking, other than for matching brackets and checking the fixed-format column rules, is done. All other options on the command-line are ignored.

/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).

/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 and /UNDEF options).

/DEFINT_KIND <value>
Sets the default KIND for type INTEGER to <value>. <value> can be 2, 3, or 4

but the value 4 is not recommended (see page 101). The default is 3, other values may not be standard conforming.

/DEFLOG_KIND <value>
Sets the default KIND for type LOGICAL to <value>. <value> can be 1, 2, or 3. The default is 3, other values may not be standard conforming.

/DEFREAL_KIND <value>
Sets the default KIND for type REAL and COMPLEX to <value>. <value> can be 1, 2, or 3. The default is 1, other values may not be standard conforming.

/DELETE_OBJ_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. /DELETE_OBJ_ON_ERROR overrides the default.

/DO1
Causes DO loops to be executed at least once to allow certain Fortran 66 programs to execute correctly.

/DREAL
Set the default KIND for REAL types to be 2 (DOUBLE PRECISION).

/DUMP
Output a variable dump in listing file.

/ERROR_NUMBERS
Error messages are accompanied by their error number.

/ERRORLOG
Creates FILE.ERR file containing a copy of all error and warning messages.

/EXPLIST [<filename>]
Generate a listing file containing a symbolic assembler listing. <filename> is the optional name of the .LIS file. If <filename> is not specified, and there is no /LIST option, the listing-file is created with the name following /LIST.

/FIXED_FORMAT
Accept source code in ISO fixed source format. This is assumed for files with .FOR or .F suffixes. If the name of the source file on the command line has no suffix and this option is supplied then .FOR is assumed to be the suffix.

/FPP
Invoke the preprocessor on all input files (this is required in order to use CIF/CELSE/CENDIF when /SPARAM and /VPARAM are not used).

/FREE_FORMAT
Accept source code in ISO free source format. If the name of the source file on the command line has a suffix which is not .FOR and not .F, this option is assumed. If the name of the source file on the command line has no suffix, .F90 is assumed.

/FULL_DEBUG
Outputs full debugging information including PARAMETERs.

/FULL_UNDEF
Like /UNDEF but INTENT(OUT) variables are set to the undefined state on entry to a subprogram and CHARACTER variables are set to the undefined state at startup. This means that an error occurs if an INTENT(OUT) or CHARACTER variable is used before being assigned a value.

/HARDFAIL
Used with /LGO to suppress entry into the debugger in the event of a run time error.

/HELP or /?
Invoke the window based help system.

/HEXPLIST [<filename>]
Like /EXPLIST but hexadecimal bytes for the instructions generated are also output.

/IGNORE <number>
Suppresses errors, warnings and comments by error number (reported by /ERROR_NUMBERS). Use this option with caution since its use can cause erroneous code to be tolerated.

/IMPLICIT_NONE
Fault on IMPLICIT type definitions.

/IMPORT_LIB <libname>
Scans a Digital Visual Fortran (DVF) generated .LIB or .DLL, automatically creating F_STDCALL declarations for all routines found in the library. Mainly for use with third party libraries compiled using DVF.

/INCLUDE <pathname>
Allows specification of list of include paths. This is appended to the list of include-paths optionally specified in the environment variable F95INCLUDE.

/ISO
Checks that the source conforms to the ISO Fortran 95 Standard.

/INTL
Sets the default KIND for type INTEGER to 3.

/INTS
Sets the default KIND for type INTEGER to 2.

/LGO
Compile, load and execute.

/LIBRARY<filename>
Specification of relocatable binary library and input files when using /LGO or /LINK.

/LINK [<filename>]
Generates an EXE file. If <filename> is not provided, the name is created by replacing the source-suffix with .EXE, e.g. with FOO.F90, the executable file would be FOO.EXE.

/LIST [<filename>]
Produces a source listing file. If <filename> is not provided, the name is created by replacing the source-suffix with .LIS, e.g. with FOO.F90, the map file would be FOO.LIS.

/LIST_INSERT_FILES
Used with /LIST to include INCLUDE files in listing.

/LOGL
Sets the default KIND for type LOGICAL to 3.

/LOGS
Sets the default KIND for type LOGICAL to 2.

/MAC_EOL
Specifies Apple Macintosh style end-of-line chracters for the source.

/MAP <filename>
Specifies file for linker map.

/MOD_PATH <pathname>
Sets alternative path for module files. More than one /MOD_PATH is permitted. This is appended on to the path contained in the optional environment variable MOD_PATH.

/NO_BANNER
Suppress compilation logos.

/NO_BINARY or /NO_CODE
Suppresses the creation of an object module.

/NO_COMMENT
Suppresses all comments, allowing only warnings and errors to be reported.

/NO_OBSOLETE
Suppress reports of obsolete and extension warning messages after the first has been displayed.

/NO_OFFSETS
Suppresses the output of address offsets in the source listing.

/NO_QUIT
Suppresses the simulated quit after a bad compilation. This normally simulates a ^C to halt a batch-file compilation. With this option set

```
IF ERRORLEVEL 1
```

must be use to test for compilation failure in a batch file if this option is set, or alternatively

```
IF NOT EXIST <file>.OBJ
```

if the /DELETE_OBJ_ON_ERROR option is also used.

/NO_SCREEN_MESSAGES
Suppresses the screen display of error, warning and comment messages when /LIST is used.

/NO_WARN_WIDE
Suppresses warnings for characters appearing in columns 133 (free format) or 73 (fixed format) and beyond in the source file.

/NO_WARN73
Old synonym for /NO_WARN_WIDE.

/NON_STANDARD
Suppresses compiler messages that warn of syntax that is permitted by the FTN95 compiler but does not conform to the Fortran 95 standard. Such warnings will be converted to errors if /IOS is used. /NON_STANDARD is switched on by default.

/OLDARRAYS
Allows array subscript checking to be used with array arguments whose corresponding dummy argument is declared with a last subscript of 1.

/OPTIMISE
Invoke the peephole and tree optimisers (/OPTIMIZE is an alternative spelling). /CHECK, /UNDEF, /DEBUG and their variants are incompatible with this option.

/OPTIONS <filename>
Specifies a file containing additional compiler options.

/P6
Uses some instructions which are only available on a Pentium Pro (P6) (or better) processor. This option cannot be used and the resulting executable will not run on a lesser machine.

/P6PRESENT
Uses /P6 only if the host machine is a Pentium Pro (or better).

/PARAMS <param-list>
Used with /LGO or /BREAK to pass options to the program rather than the compiler. This must be the last FTN95 option on the command line.

/PENTIUM
Outputs code for Pentium or above processors (produces a slower executable on 386 or 486).

/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
Inserts code to record how many times each line is executed.

/PROGRESS_METER
Adds to the console title bar an estimate of how far the compilation has progressed when compilation is estimated to take longer than about one second.

/QUICK_BOUNDS
Outputs code to do a quick check on array bounds. Only checks total bound size. This gives some protection against array-bounds being exceeded without the run-time expense of /CHECK.

/RESTRICT_SYNTAX
Increases compiler strictness in order to reject 'dubious' code such as the use of an EQUIVALENCEd variable as a DO-loop index.

/RUNTRACE
Adds a call to RUN_TRACE@ after every executable line. This is declared as:

```
SUBROUTINE RUN_TRACE@(LNUMBER, FUNCT_NAME, FILE_NAME)
INTEGER(3) LNUMBER
CHARACTER(*) FUNCT_NAME, FILE_NAME
```

A default routine is supplied which simply outputs these arguments preceded by "**** TRACE ", but you can supply your own version of this routine.

/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.

/SEARCH_INCLUDE_FILES
Searches through INCLUDE files for CONTAINS. Switch this on only if an INCLUDE file contains executable statements. This option is set by /ISO. When compiling typical Fortran 77 programs containing many INCLUDE statements, this will speed compilation.

/SET_ERROR_LEVEL <code> <number>

<number> is a value reported by /ERROR_NUMBERS. <code> is one of:

Suppress	Ignore the condition.
Comment	Change the condition to a comment.
Warning	Change the condition to a warning.
Restricted	Change the condition to an error if /RESTRICT_SYNTAX is used otherwise suppress.
Error	Change the condition to an error.
Fatal	Change the condition to a fatal error (aborts compilation).

For example,

```
/SET_ERROR_LEVEL Error 298
```

would cause the condition leading to the warning "Variable X has been used without being given a value", to be upgraded to an error condition. Clearly it is risky to downgrade a condition. Only the first character of <code> is significant.

/SILENT

Suppress warning and comment messages. Also when /SILENT is used, the message that is output on the screen at the end of the compilation of a program unit does not include the numbers of warning and comment messages.

/SINGLE_LINE

Only prints the first line of an error message, ignoring any continuations. This is ignored if /BRIEF is specified. Without this option, when an error is reported, the offending line is output together with any continued lines.

/SPARAM <integer>

Sets value of SPECIAL PARAMATERS (see page 125).

/STATISTICS

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

/SUPPRESS_ARG_CHECK

Suppresses /CHECK mode checking for argument constistency across routines.

/TIMING

Plants code to do per-routine timing.

/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.

/UNLIMITED_ERRORS

Carry on compiling even after 12 error messages.

/VERSION
Display version information.

/VPARAM <name> <integer>
Like /SPARAM but allows the user to define a SPECIAL PARAMETER by name, instead of having all the same value (see page 125).

/WARN_FATAL
All warnings and comments become errors.

/WIDE_SOURCE
Line-truncation at the 132nd (or 72nd) column is suppressed.

/WINDOWS
Program is compiled for use with ClearWin+.

/XREAL
Sets the default KIND for REAL types to be 3.

/XREF [<filename>]
Generates a cross-reference .XRF file. If <filename> is specified, the file name is created by replacing the source file-name's suffix with .XRF, e.g. with FOO.F90, the resulting name would be FOO.XRF.

/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. 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 /CONFIGURE indicates which options can be configured and hence have an opposite.

5.

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 95/98 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, FTN95 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 2.

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:

  ```
  FTN95 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
  ```

When using Plato, select “Checkmate enabled” in the File Options dialog box. After building your executable, click on the Debugger button on the toolbar.

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.

Command line switches for SDBG

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.

Passing run-time command line parameters

If the executable that you are debugging uses command line parameters, then you will need to pass these to SDBG. To do this use /PARAMS (or /P) followed by a space and then the list of parameters to pass. Everything to the right of this option is passed on the command line to the program being debugged.

Hiding routines on the call stack

By default, Salford library routines (defined in *salflibc.dll*) are not shown in the SDBG call stack. The SDBG command line option /NO_HIDE_SALFLIBC (or /NHS) forces all Salford library routines to be shown.

Hiding the debugger

The SDBG option /SILENT causes the debugger to be hidden unless there is a run-time error.

SDBG version number

The SDBG option /VERSION (or /V), causes SDBG to output the version number of the debugger and then exit.

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,

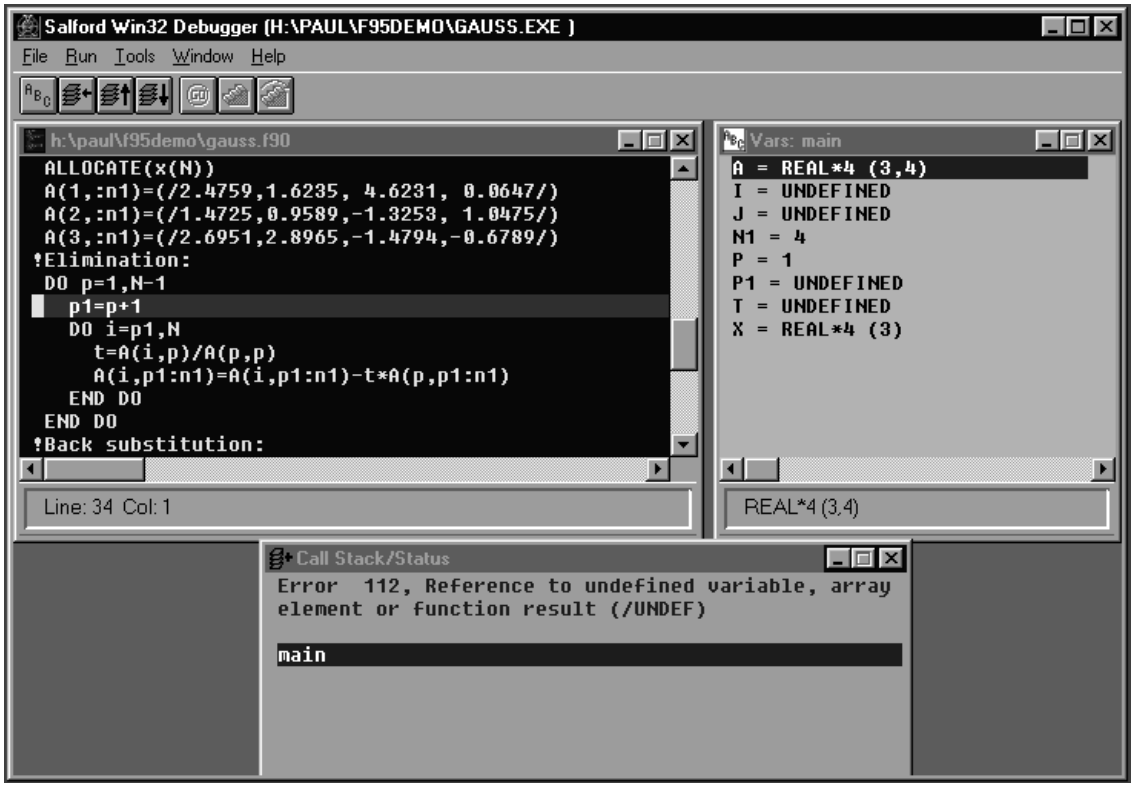


Figure 1. An screen shot from SDBG.

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 *eps* is greater than 1	1	1	*eps*>1
Stop when *eps* is greater than 1 and I know this is after the 654th iteration	654	1	*eps*>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 53).

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 51).

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 51).

More than one source file can be compiled with /PROFILE (FTN95 only). 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 95 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 42 for a source code window, immediately above for a variables window, and on page 49 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 *ptr*, this would produce a memory dump showing the physical memory used to store *ptr*.
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 52).
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 52).
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 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 52).
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 `DOINIT`

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

`let i=10`	Simple variable assignment
`let arr(j)=arr(count)`	Array element assignment
`let shape.colour=0x0f3229`	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.

Programming data breakpoints

The following routines can be used to set data breakpoints within a program. These provide much greater flexibility than static breakpoints. Because in Fortran arguments are passed by reference, the argument below that is named ADDRESS is simply the name of a variable or an element of an array. Only one data breakpoint

can be active (via these routines) at any time. This means that a call to any one of these routines overwrites the previous data breakpoint setting.

These routines are for use with SDBG, however SET_READ_WRITE_BREAK@ and SET_WRITE_BREAK@ will work in DOS mode under DBOS.

```
SUBROUTINE SET_WRITE_BREAK@(ADDRESS)
INTEGER*4 ADDRESS
```

This routine sets the data breakpoint register to ADDRESS and covers 4 bytes. For example:

```
INTEGER*4 IARRAY(10)
CALL SET_WRITE_BREAK@(IARRAY(5))

DO I=1,10
  IARRAY(I)=I
ENDDO
```

will stop in SDBG when I gets to 5 and an attempt is made to write to IARRAY(5).

```
SUBROUTINE SET_READ_WRITE_BREAK@(ADDRESS)
INTEGER*4 ADDRESS
```

This routine is identical to SET_WRITE_BREAK@ except that a breakpoint will occur if the memory is read or written (also referred to as a USE breakpoint).

```
SUBROUTINE SET_COUNTED_WRITE_BREAK@(ADDRESS,COUNT)
INTEGER*4 ADDRESS,COUNT
```

This routine is similar to SET_WRITE_BREAK@ except that the break will only occur after the memory location has been written to COUNT times.

```
SUBROUTINE SET_MASKED_WRITE_BREAK@(ADDRESS,MASK,VALUE)
INTEGER*4 ADDRESS,MASK,VALUE
```

This routine is used to trigger a break when the value obtained by ANDing the INTEGER*4 at ADDRESS with MASK is equal to VALUE. If MASK=(-1), a break is triggered when the value at ADDRESS is equal to VALUE.

6.

Program development

Diagnostic facilities

FTN95 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 Fortran 95 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 /ISO option is used otherwise they are classed as warnings.

2) WARNINGS are output for one of two reasons:

   - If the program is correct Fortran 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 95 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 95.
     ```

   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 7) 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:

```
FTN95 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.

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 95 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 18) 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 FTN95 compile-time options /CHECK, /UNDEF or /QUICK_BOUNDS. These keywords may also appear as part of an OPTIONS directive.

For example:

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

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 FTN95:

INTEGER (KIND=1)	-128 to +127
INTEGER (KIND=2)	-32768 to +32767
INTEGER (KIND=3)	-2147483648 to +2147483647
REAL (KIND=1)	±(1E-37 to 1e+39) (approx.)

DOUBLE PRECISON	±(1D-307 to 1D+309) (approx.)

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

If a CHECK 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 5). 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 4).

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 I=1,100
  CALL FRED(I)
  . . .
ENDDO
. . .
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 also checked. 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 95 (although it is valid Fortran 66).

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.

Note: When using Fortran 95 assumed-shape arrays in an external subprogram, you must provide a corresponding INTERFACE statement where the subprogram is called.

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.

Checking for undefined variables (/UNDEF)

/UNDEF (which implies /CHECK) causes FTN95 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 ISO standard intrinsic function such as SIN, COS etc.,
- as the expression used within a logical or arithmetic IF statement.

/FULL_UNDEF adds checks for character assignments and concatenations to /UNDEF as well ensuring that INTENT(OUT) subprogram arguments are not used before being assigned a value.

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 Chapter 5. An undefined integer has one of the following values:

INTEGER (KIND=1)	-128
INTEGER (KIND=2)	-32640
INTEGER (KIND=3)	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.

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 is one possible source of error that may arise when using a substring reference of the form A(I:J), namely that 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 segment would cause a run-time error:

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

Using ALLOCATE

A Fortran program has access to two areas of memory. The *stack* is automatically used by the compiler for the storage of all local objects. The *heap* is used for memory that is made available via ALLOCATE statements in the program (or via the Salford GET_STORAGE@ routine). When a program is running with at least part of the code compiled in CHECK mode, there are in fact two heaps, one for CHECK mode data, the other for normal data.

The primary concern when making extensive use of the ALLOCATE statement is to avoid memory *leakage*. In other words, to ensure that every ALLOCATE is matched by a corresponding DEALLOCATE. Note however that a) all ALLOCATEd memory is released when the program terminates, and b) in Fortran 95, DEALLOCATE is automatically called when the corresponding pointer *goes out of scope*. That is, if a pointer is local to a sub program and is not SAVEd, then DEALLOCATE is automatically called when control is returned from the sub program. FTN95 is like all other compilers in that it does not check for memory leakage. If you make extensive use of the ALLOCATE statement then you should consider providing your own checking mechanism. One simple device is to increase a counter each time ALLOCATE is called and to decrease it for each call to DEALLOCATE, checking the value of this counter when the program terminates.

A secondary concern is to avoid *dangling* pointers. A dangling pointer is a pointer that is used by mistake after a call to its corresponding DEALLOCATE. The following paragraphs explain how to check for dangling pointers.

The default size of the heap when using CHECK mode is half of the total physical memory (RAM). Using a large heap increases the chances of detecting dangling pointers. This is because the memory in a smaller heap is more likely to be reused with the result that a dangling pointer can be used even though it is now pointing to the wrong object.

Whilst developing your program, you can make sure that dangling pointers are detected by preventing the heap from being re-used with the following call.

```
CALL SET_CHECK_REUSE@(0)
```

This may cause a program to fail with an "Out of Memory" error because memory released by DEALLOCATE will not be reusable. However, you can also increase the size of the heap (see below) until your program runs. This may lead to excessive paging to disc but will guarantee that dangling pointer errors are detected. Once a program has been tested, all run-time checking options should be removed in which case a call to this routine will have no effect.

When memory is ALLOCATEd for arrays above a certain size, the normal heap is used instead of the CHECK mode heap. The critical array size that triggers this spill-over effect can be set with the following call:

```
CALL SET_BIG_CHUNK_SIZE@(size)
```

Setting the size to HUGE(1) suppresses the spill-over mechanism.

You can specify the size of the CHECK mode heap by using the environment variable FTN95CHECKHEAP. For example,

```
SET FTN95CHECKHEAP=32
```

sets the size of CHECK mode heap to 32 Mb. The larger this value, the better the chance of detecting a dangling pointer, but this value should not be set too high in order to avoid excessive paging to disc.

The size of the CHECK mode heap can also be set within a program by calling the routine SET_CHECK_HEAP@. This takes a single integer argument that specifies the size of the heap in bytes. If this routine is called then the call must occur before the first ALLOCATE statement in the program. Use of this routine after an ALLOCATE statement will cause a run-time error. This value over-rides a value set using the environment variable FTN95CHECKHEAP. The size of the CHECK mode heap becomes irrelevant when the command line checking options are not used.

The minimum size for the CHECK mode heap is 1 Kb although it is recommended this be set to at least 1 Mb. However, using a small heap size will prevent the compiler from detecting dangling pointers. Here is some sample code that illustrates this feature:

```
PROGRAM test
INTEGER, ALLOCATABLE :: a(:)
INTEGER, POINTER :: p(:)
INTEGER :: i
CALL SET_CHECK_HEAP@(8*1024*1024)
DO i = 1, 4
  ALLOCATE(a(1000000))
  IF (i == 1) p => a
  IF (i /= 3) DEALLOCATE(a)
END DO
p = 1 ! Dangling pointer error
END
```

The dangling pointer error in this program is not trapped. However, it is detected if the heap size is increased from 8 to 16 Mb (assuming the program is compiled using CHECK mode).

7.

Optimisation and efficient use of Fortran

Introduction

This chapter describes the FTN95 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 (or /OPTIMIZE) selects the optimisation facility described below.

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.

Using a coprocessor

The compiler will automatically generate correct code for an Intel compatible numeric coprocessor.

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:

```
K = I + J
L = K * I
```

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 susequent calculation.

Note that this floating point register tracking is not performed when the /DEBUG compiler option is used or implied.

- 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
```

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.

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 occuring. 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 CALLs 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 FTN95's compilation diagnostics).

Efficient use of Fortran

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/=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==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`.

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.

- AND, OR, XOR, NOT, LS, RS, LR, RR, SHFT, LT, RT, LOC, CCORE1, CORE1, CORE2, CORE4, FCORE4 and DCORE8.
  Note: these functions are FTN95 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.
- SQRT.
- SIN, COS and TAN .
- 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(LEN=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 I=1,100
  A(I:I) = ' '
END DO
. . .
```

Format statements

Unlike many Fortran implementations, FTN95 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(LEN=10)::F
F = '(3F10.4)'
```

```
. . .
WRITE (2,F)X,Y,Z
```

It could be rewritten as follows:

```
CHARACTER(LEN=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.

Efficient use of Fortran 90

The following notes relate to features introduced into the Fortran 90 standard.

- FTN95 produces faster code with ALLOCATABLE arrays as against POINTER arrays because the ALLOCATABLE attribute guarantees contiguous storage and no aliasing.
- Temporary arrays (required only during the life time of a function call) are better implemented using automatic arrays in preference to ALLOCATABLE arrays because these can be allocated more efficiently.
- Fixed-size arrays are more efficient than variable-sized arrays so, when the logic of the program and memory constraints allow, fixed-size arrays are preferred.
- Specifying the INTENT attribute of a dummy argument improves diagnostics and helps the optimiser and code generator.
- The use of ENTRY and EQUIVALENCE prevents certain optimisations from taking place.

8.

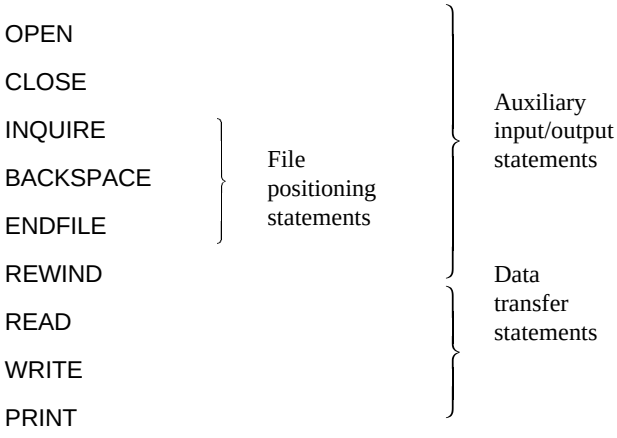
Fortran input/output

Overview

The FTN95 input/output statements allow ISO 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:



There is also one format descriptor statement :

FORMAT

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=2,END=20) A
```

The general form of these specifiers will be obvious from these examples. The full details of the input/output statements in Fortran 95 are given, for example, in the *Fortran 95 Handbook* (Adams, Brainerd, Martin, Smith and Wagener. The MIT Press). This chapter contains a quick reference guide to the input/output and device control statements. The value of strings given in the following list assumes that all leading spaces and nulls have been removed. The name <variable> refers to a numeric or string variable as appropriate.

Note: The standard Fortran I/O system is not compatible with the FTN95 library file-manipulation routines described in chapter 25. These functions provide low-level file access.

OPEN

It is possible to open a file dynamically (that is, at run time) by means of the Fortran 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.

The general form of the OPEN statement is :

```
OPEN(<olist>)
```

where <olist> is a list of specifiers from the list below:

UNIT= <value>
: <value> is the unit number. The keyword and equals sign are optional if this is the first item in the list.

FILE= <filename>

<filename> is the name of the file to which the unit will be connected when it is opened. This is only optional if the specifier STATUS=SCRATCH is present.

IOSTAT= <variable>

<variable> is an integer numeric variable into which the input/output status will be written. The variable will be positive if an error occurred, otherwise it will be zero.

ERR= <label>

<label> is a label to jump to in the event of an error.

ACCESS= <string>

This specifies the type of access for which the file is to be opened. <string> is either DIRECT or SEQUENTIAL.

FORM=<string>

<string> is FORMATTED, UNFORMATTED or PRINTER.

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.

ACTION= <string>

<string> specifies that READ, WRITE or READWRITE operations will be allowed on the file. The default is READWRITE.

BLANK= <string>

This option may only appear for files being connected for formatted input/output. <string> is a string which when the trailing spaces are removed is either NULL or ZERO. The default is NULL. If NULL, then all spaces in formatted numeric fields are ignored, if ZERO then spaces in formatted numeric fields are treated as zeros.

DELIM= <string>

<string> specifies the delimiter to be used in character constants. It is always one of APOSTROPHE , QUOTE or NONE. The default is NONE.

PAD= <string>

<string> specifies whether input records are space padded. The value is either YES or NO. The default is YES.

POSITION= <string>

<string> is a character expression that, when trailing spaces have been removed, is ASIS , REWIND or APPEND.

RECL= <value>

<value> is an integer expression, which gives record length. For direct access files, this specifier is obligatory.

STATUS= <string>

<string> is a character expression which, when trailing spaces are removed, gives the value OLD, NEW, REPLACE, SCRATCH or UNKNOWN.

The following options for <string> are not in the ANSI Standard:

'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.

'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.

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), 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.

<olist> must contain exactly one external unit specifier <unit> and may contain, at most, one of each of the other specifiers.

CLOSE

The CLOSE statement is used to disconnect a file which has been connected with OPEN. It has the following specifiers associated with it:

UNIT= <value>

<value> is the unit number. The keyword and equals sign are optional if this is the first specifier in the list.

IOSTAT= <variable>
<variable> specifies an integer variable into which the input/output status will be written. A positive value will be written on error, zero otherwise.

ERR= <label>
<label> specifies a label to jump to in the event of an error.

STATUS= <string>
<string> is either KEEP or DELETE. The default for named files is KEEP, for scratch files is DELETE.

INQUIRE -- file or unit

The INQUIRE statement is used 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 of the statement. It has the following specifiers associated with it:

UNIT= <value>
<value> specifies the unit number for INQUIRE by unit. The keyword and equals sign are optional if this is the first specifier in the list.

FILE= <string>
<string> specifies the filename for INQUIRE by name. UNIT must not be used if is used and vice-versa.

IOSTAT= <variable>
<variable> specifies an integer variable into which the input/output status may be written.

ERR= <label>
<label> is a label to jump to in the event of an error.

EXIST= <variable>
<variable> specifies a logical variable that will be set true if the file or unit exists, otherwise false.

OPENED= <variable>
<variable> specifes a logical variable which will be set true if a connection exists to the file or logical unit number, otherwise false.

NUMBER= <variable>
<variable> is the variable into which the unit number attached to the file is written, or -1 if the file is not attached to a unit.

NAMED= <variable>
<variable> is set to true if the specified file is named, otherwise false.

NAME= <variable>
<variable> will contain the name of the file (if named) or undefined.

ACCESS= <variable>
<variable> will contain the access type for which the unit or file is opened. This will be SEQUENTIAL, DIRECT or UNDEFINED.

SEQUENTIAL= <variable>
<variable> will contain YES, NO or UNKNOWN depending on whether the file or unit is opened for sequential access.

DIRECT= <variable>
<variable> will contain YES, NO or UNKNOWN depending on whether the file or unit is opened for direct access.

FORM= <variable>
<variable> will contain FORMATTED or UNFORMATTED depending on the connection to the file or unit.

FORMATTED= <variable>
<variable> will contain YES, NO or UNKNOWN depending on whether the file or unit is opened for formatted file operations.

UNFORMATTED= <variable>
<variable> will contain YES, NO or UNKNOWN depending on whether the file or unit is opened for unformatted file operations.

RECL= <variable>
<variable> will contain the maximum record length for a formatted or unformatted file, or undefined if there is no connection.

NEXTREC= <variable>
In direct access files, <variable> will contain one more than the last record number read (records start at 1) or undefined if the position is not known.

BLANK= <variable>
<variable> will contain NULL, ZERO or UNDEFINED, depending on the connection to the unit number or file.

POSITION= <variable>
<variable> will contain REWIND, APPEND, ASIS or UNDEFINED depending on the connection to the unit or file.

ACTION= <variable>
<variable> will contain READ, WRITE, READWRITE or UNDEFINED depending on the connection to the unit or file.

READ= <variable>
<variable> will contain YES, NO or UNKNOWN depending on the connection to the unit or file.

WRITE= <variable>
<variable> will contain YES, NO or UNKNOWN depending on the connection to the unit or file.

READWRITE= <variable>
<variable> will contain YES, NO or UNKNOWN depending on the connection to the unit or file.

DELIM= <variable>
<variable> will contain APOSTROPHE, QUOTE, NONE or UNDEFINED depending on the connection to the unit or file.

PAD= <variable>
<variable> will contain YES or NO depending on the connection to the unit or file.

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@.

INQUIRE -- input/output list

The second use of INQUIRE is to find the length of an unformatted input/output list. When used for this purpose, INQUIRE has the following single specifier:

IOLENGTH= <variable>
<variable> will contain the length of an unformatted input/output list.

BACKSPACE

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.

The following specifiers are associated with the BACKSPACE statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if this is the first item in the list. If unit number is the only argument, the brackets around the argument list are optional.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error.

ERR= <label>
<label> specifies a label to go to in the event of an error.

REWIND

Execution of a REWIND statement causes the specified disc file to be positioned at its initial point.

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if this is the first item in the list. If unit number is the only argument, the brackets around the argument list are also optional.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error.

ERR= <label>
<label> specifies a label to go to in the event of an error.

ENDFILE

Execution of an ENDFILE statement causes an end-of-file (EOF) record to be written to the specified disc file.

UNIT= <value>
<value> specifies the unit number. The keyword is optional if it is the first item in the list. If the unit number is the only argument, the brackets around the argument list are optional.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error.

ERR= <label>
<label> specifies a label to go to in the event of an error.

READ -- sequential formatted

This section describes the use of the READ command, for sequential formatted connections to files.

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional, if this is the first item in the list of specifiers.

FMT= <string, label or *>
Specifies the format in which the data will be read. The keyword is optional if this is the second item in the list of specifiers.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

END= <label>
<label> specifies a label to go to on end of file.

ADVANCE= <string>
<string> should be set to NO for non-advancing input, YES or absent for advancing input.

SIZE= <variable>
<variable> will contain the number of characters actually read (non-advancing only).

EOR= <label>
<label> specifies a label to go to on end of record (non-advancing).

READ -- internal

The READ statement can be used to input data from internal files. When used for this purpose, the specifiers associated with it are as follows:

UNIT= <string>
<string> is a character variable. The keyword and equals sign are optional if it is the first item in the list.

FMT= <string, label or *>
Specifies the format in which the data will be read. The keyword and equals sign are optional if this is the second item in the list of specifiers.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

END= <label>
<label> specifies a label to go to on end of file.

READ -- namelist

Read a file using a namelist.

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the specifier list.

NML= <namelist-group-name>
Namelist group. The keyword and equals sign are optional if it is the second item in the specifier list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

END= <label>
<label> specifies a label to go to on end of file.

READ -- sequential unformatted

When READ is used to read data from sequential unformatted files, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the specifier list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

END= <label>
<label> specifies a label to go to on end of file.

READ -- direct formatted

When READ is used to read data from direct formatted files, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the specifier list.

FMT= <string or label>
Specifies the format for the data to be read in. The keyword and equals sign are optional if this is the second item in the list.

REC= <value>
<value> specifies the record number to read from.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

READ -- direct unformatted

When READ is used to read data from direct unformatted files, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if this is the first item in the list.

REC= <value>
<value> specifies the record number to read from.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

WRITE -- sequential formatted

When WRITE is used to write data to a sequential formatted file, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if this is the first item in the list.

FMT= <string, label or *>
Specifies the format for the data to be written. The keyword and equals sign are optional if this is the second item in the list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

ADVANCE= <string>
<strings> equals NO for non-advancing output, YES or absent for advancing output.

WRITE -- internal

When WRITE is used to write data to an internal file, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the list.

FMT= <string, label or *>
Specifies the format for the data to be written. The keyword and equals sign are optional if it is the second item in the list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

WRITE -- namelist

Write a file using a namelist.

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the list.

NML= <namelist-group-name>
Namelist group. The keyword and equals sign are optional if this is the second item in the list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

WRITE -- sequential unformatted

When WRITE is used to write data to a sequential unformatted file, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the list.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

WRITE -- direct formatted

When WRITE is used to write data to a direct formatted file, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the list.

FMT= <string or label>
Specifes the format for the data to be written. The keyword and equals sign are optional if it is the second item in the list.

REC= <value>
<value> specifies the record number to start writing to.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

WRITE -- direct unformatted

When WRITE is used to write data to a direct unformatted file, the following specifiers are associated with the statement:

UNIT= <value>
<value> specifies the unit number. The keyword and equals sign are optional if it is the first item in the list.

FMT= <string or label>
Specifies the format for the data to be written. The keyword and equals sign are optional if it is the second item in the list.

REC= <value>
<value> specifies the record number to start writing to.

IOSTAT= <variable>
<variable> will contain zero if the operation was successful, otherwise a positive value to indicate an error, -1 to indicate EOF or -2 to indicate EOR.

ERR= <label>
<label> specifies a label to go to in the event of an error.

FORMAT edit descriptors

The following specifiers are edit descriptors that can be used with the FORMAT statement. Generally, w represents an integer specifying the field width and m represents an integer specifying the number of digits to be displayed. Under Fortran 95, if w is set to zero, then the field width is automatically set to the minimum value that can be used.

I <w.m>
Integer

B <w.m>
Integer as binary

O <w.m>
Integer as octal

Z <w.m>
Integer as hexadecimal

F <w.d>
Real, fixed form

E <w.dEe>
Real, exponent form

EN <w.dEe>
Engineering notation (exponent is 0 or divisible by 3)

ES <w.dEe>
Scientific notation, non-zero significands are greater than or equal to 1.0 and less than 10.0 (see note below)

G <w.dEe>
General, for any intrinsic data type

L <w>
Logical

A <w>
Character

D <w.d>
Real (D used as exponent letter)

<w>H
Hollerith (obsolescent feature, should be avoided)

Note ESw.d edit descriptors are explicitly restricted by the standard to numbers where the absolute value of the decimal exponent is less than 1000. If the decimal exponent of the number is outside of this range, then the decimal exponent is divided by 10 and preceded by a dollar sign "$" if positive and an equals sign "=" if negative. For example, if $1e^{2000}$ were written using ES12.3, the result would be "xxx1.000$200" where x is a space. If $1e^{-2000}$ were written using ES12.3, the result would be "xxx1.000=200" where x is a space. To avoid this situation use the ESw.dEe edit descriptor.

FORMAT control descriptors

The following specifiers are control descriptors that can be used with the FORMAT statement :

T<n>
Tab to position n, positions start at 1

TL<n>
Tab left n

TR<n>
Tab right n

<n>X
n spaces

<n>/
n new records. n can be omitted. The default value for n is 1.

S
Use default of SS and SP

SS
Suppress leading + signs

SP
Print leading + signs

<k> P
Set scale factor for E, F and G processing

BN
Interpret spaces as nulls

BZ
Interpret spaces as zeros

Business Editing

FTN95 business editing is an extension to the standard that 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

For example:

```
WRITE(*,"(B'####')") 147
```

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 preceed 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 8-1 below illustrate the use of the B edit descriptor.

Other Extensions

"Line-printer" carriage control characters

The Fortran standard defines the following:

+	remain on the same line (overprint)
0	double line space
1	advance to the beginning of the next page

In addition, FTN95 uses:

- (minus)	double line space

Printing to a device

FTN95 permits you to write directly to a given device. For example,

```
OPEN(UNIT=8, FILE='LPT1:')
WRITE(8, '(A)') 'Output to printer'
CLOSE(8)
```

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'$ZZZZZZ#'	$ 99
99	B'$$$$$$$#'	$99
308126	B'$***,***,**#.##'	$****308,126.00

Table 8-1 Business editing examples

9.

Kind parameters for intrinsic types

This chapter lists the KINDs associated with each of the intrinsic types, and some information about each KIND.

Logical KINDs

Logical	Default	8 bit	16 bit	32 bit
Kind number	3	1	2	3

Integer KINDs

Integer	Default	int8	int16	int32	int64
Kind number	3	1	2	3	4
Binary digits	31	7	15	31	63
Radix	2	2	2	2	2
Range	9	2	4	9	18
Huge	2147483647	127	32767	2147483647	2^{63}-1
Bit size	32	8	16	32	64

INTEGER(KIND=4) is a 64-bit integer type. Input and output of these large integers is fully supported. However, as the x86 architecture only provides native support for this data type using the FPU, programs will be considerably slower using this type when compared with INTEGER(KIND=3). Note that the compiler's memory limitations are still 32-bit (e.g. array indices). Note also that several components of the run-time system (e.g. the IOSTAT=var, INQUIRE(...)) will not work correctly

when INTEGER(4) arguments are given to them. Because of this the use of the compiler command line switch "/DEFINT_KIND 4" is not recommended unless there is only simple (or no) I/O.

Real KINDs

Real	Default	Single	Double	Extended
Kind number	1	1	2	3
Binary digits	24	24	53	64
Decimal digits*	7	7	16	19
Max exponent	128	128	1024	16384
Min exponent	-125	-125	-1021	-16381
Precision	6	6	15	18
Radix	2	2	2	2
Range	37	37	307	4931
Epsilon	1.1920929E-07	1.1920929E-07	2.2204460E-16	1.0842022E-19
Tiny	1.1754944E-38	1.1754944E-38	2.2250739E-308	3.3621031E-4932
Huge	3.4028235E+38	3.4028235E+38	1.7976931E+308	1.1897315E+4932

* Approximate number of significant decimal figures.

Complex KINDs

Complex	Default	Single	Double	Extended
Kind number	1	1	2	3
Precision	6	6	15	18
Range	37	37	307	4931

10.

Using modules

Modules in Fortran 95

The concept of modules is a powerful feature that was introduced into the Fortran 90 standard. Modules can be used for many purposes, from sharing data between subroutines and functions (somewhat in the manner of COMMON, but in a cleaner, more abstract way), to providing support for user-defined datatypes and operations thereon.

The first part of this chapter provides a brief introduction to the syntax of module definitions and references by means of some examples. This is one of the few language-related areas expanded to this level in this manual and is intended as a supplement to the description in the accompanying textbook. This is because this topic is perhaps the most important aspect of the new facilities added to Fortran 77 by the Fortran 90 standard and many of the ideas will be somewhat unfamiliar to some Fortran 77 programmers.

The second part of this chapter discusses how these ideas are implemented, in terms of what happens when a MODULE is compiled under FTN95.

Using modules for global data

Modules provide an alternative method to using COMMON blocks for sharing data between different program units without passing such data as arguments. While the end result is similar, the two achieve this effect in different ways. COMMON relies on “storage association” -- a COMMON block definition specifies the layout of a region of storage, and since other COMMON definitions refer to the same region of storage, different routines can access the same data. It is left up to the programmer to check that the common definitions match. Many programmers make use of the INCLUDE facility (non-standard in Fortran 77) to ensure that the same definition of

the common block is used in every place. There is no mechanism for ensuring that a particular piece of storage in the common block is not used as an integer in one place and a floating point quantity in another (although a number of static analysis tools are available which will perform this kind of checking).

By contrast, the way that modules share data is by "name association". Each of the names accessible from a program unit referencing the module is made known together with its type, so that the referencing module refers to a central definition of the data contained in the module. This removes the possibility of conflicting declarations.

An example of a module used in this way follows:

```
MODULE DATA_MODULE
    REAL :: ARRAY1(10), B, ARRAY2(20,20)
    INTEGER :: COUNT=0
    INTEGER, PARAMETER :: ORDER=10
    COMPLEX C_ARRAY(ORDER,ORDER)
END MODULE DATA_MODULE
```

The module can be referenced with a USE statement in the program unit which requires access to the global data. USE statements must come after the subprogram header statement (i.e. PROGRAM, SUBROUTINE, FUNCTION etc.) and before the local declarations for the program unit. The syntax of the USE statement provides some flexibility, allowing only specified entities to be accessed, or allowing certain entities to be used under a different name from their declaration in the module.

Here are some examples based on the module definition above. To allow you to reference all the objects in the module without qualification:

```
USE DATA_MODULE
```

To allow you to reference only the arrays, use:

```
USE DATA_MODULE, ONLY: ARRAY1, ARRAY2, C_ARRAY
```

It is possible to rename some objects defined in the module. This allows local objects with the same names as objects in modules to be defined. In this example ARRAY1 and ARRAY2 are renamed to be referred to as VECT and MATRIX:

```
USE DATA_MODULE,  VECT => ARRAY1, MATRIX => ARRAY2
```

Modules for data abstraction

As well as containing data declarations, modules can contain subroutine and function definitions which provide operations on the data. Used in conjunction with the PRIVATE attribute, to make particular data objects not directly accessible from

outside the module, this provides a mechanism for construction of aggregate types and a set of operations which are allowed on them.

The following example implements some operations on integer sets:

```
MODULE INTEGER_SETS
INTEGER, PARAMETER :: MAX_CARD = 100

TYPE SET
  PRIVATE
  INTEGER CARD
  INTEGER MEMBERS(MAX_CARD)
END TYPE SET

INTERFACE OPERATOR (.IN.)
  MODULE PROCEDURE MEMBER_OF
END INTERFACE

INTERFACE OPERATOR (*)
  MODULE PROCEDURE INTERSECT
END INTERFACE

CONTAINS

INTEGER FUNCTION CARD(S)
  TYPE (SET) S
CARD = S%CARD
END FUNCTION CARD

LOGICAL FUNCTION MEMBER_OF(X, S)
  INTEGER X
  TYPE (SET) S
  MEMBER_OF = ANY(S % MEMBERS(1 : S%CARD) .EQ. X)
END FUNCTION MEMBER_OF

TYPE (SET) FUNCTION INTERSECT(S1, S2)
  TYPE (SET) S1, S2
  INTEGER I
  INTERSECT % CARD = 0
  DO I = 1, S1%CARD
    IF ( S1%MEMBERS(I) .IN. S2 ) THEN
      INTERSECT%CARD = INTERSECT%CARD + 1
      INTERSECT%MEMBERS(INTERSECT%CARD)=S1%MEMBERS(I)
    END IF
  END DO
```

```
END FUNCTION INTERSECT

TYPE (SET) FUNCTION BUILD_SET(V)
  INTEGER V(:)
  INTEGER J
  BUILD_SET%CARD = 0
  DO J = 1, SIZE(V)
    IF (.NOT.(V(J) .IN. BUILD_SET )) THEN
      IF (BUILD_SET%CARD < MAX_CARD) THEN
        BUILD_SET%CARD = BUILD_SET%CARD + 1
        BUILD_SET%MEMBERS(BUILD_SET%CARD) = V(J)
      ELSE
        PRINT *,'Maximum set size exceeded'
        STOP
      END IF
    END IF
  END DO
END FUNCTION BUILD_SET

FUNCTION VECTOR(S)
  INTEGER, POINTER :: VECTOR(:)
  TYPE (SET) S
  INTEGER I, J, K
  ALLOCATE( VECTOR(S%CARD) )
  VECTOR = S%MEMBERS(1:S%CARD)
  DO I = 1, S%CARD - 1
    DO J = I + 1, S%CARD
      IF (VECTOR(I) > VECTOR(J) ) THEN
        K = VECTOR(J)
        VECTOR(J) = VECTOR(I)
        VECTOR(I) = K
      END IF
    END DO
  END DO
END FUNCTION VECTOR

END MODULE INTEGER_SETS
```

There are a number of points to note here:

- The module defines a type SET, which can be used to declare objects in the program unit referencing the module. However, the components of the type are PRIVATE, which means they can only be accessed by procedures within the module. The referencing routine thus has no dependence on precisely how the type is implemented. In the above example the set is implemented by a fixed

length array, but this could be changed to a scheme where the set is represented by a linked list, without the code which uses the module needing to be recompiled.

- An operator ".IN." has been defined. This returns logical.TRUE. or .FALSE. depending on whether the integer specified by the left operand is a member of the set specified by the right operand. The ".IN." operator is used within the module itself in two places.
- The "*" operator is *overloaded* with the operation of set intersection. This means that applying the "*" operator to two sets will return another set which is the intersection of the two. Of course, use of "*" with two integers still has the meaning of multiplication. Attempting to use "*" with an integer and a set for example will produce a compilation error "No specific match for reference to generic *".
- The procedure CARD defined in the module is accessible to the referencing routine. Note that a procedure to extract the cardinality of the set is necessary, since the component CARD of the SET type is not directly accessible outside the module.

A trivial example illustrating use of this module follows:

```
PROGRAM SET_TEST
USE INTEGER_SETS

TYPE (SET) S1,S2

S1 = BUILD_SET( (/ 7,3,1,8,4,5 /) )
S2 = BUILD_SET( (/ 2,4,6,8 /) )

PRINT *, 'S1   : ', CARD(S1),    ' ELEMENTS: ',VECTOR(S1)
PRINT *, 'S2   : ', CARD(S2),    ' ELEMENTS: ',VECTOR(S2)
PRINT *, 'S1*S2: ', CARD(S1*S2),' ELEMENTS: ',VECTOR(S1*S2)

END PROGRAM SET_TEST
```

This produces the output:

```
S1   :  6  ELEMENTS:  1 3 4 5 7 8
S2   :  4  ELEMENTS:  2 4 6 8
S1*S2:  2  ELEMENTS:  4 8
```

Modules and FTN95

When looking for a module, say "LONG_NAME", FTN95 will look for a file called "LONG_NAME.MOD" if it is a Win32 compiler, or "LONG_NAM.MOD" with a DBOS compiler. If this file cannot be found in the current directory, FTN95 will search the *module-path*. The module-path is constructed from the environment variable MOD_PATH together with any supplied /MOD_PATH arguments appended (/MOD_PATH is a compiler command line option). When using a DBOS compiler, if the MOD file is still not found, FTN95 searches for files named using the sequence below:

```
LONG_NAM.@00,  ...,  LONG_NAM.@09,  LONG_NAM.@0A,  ...,
LONG_NAME.@0Z, LONG_NAM.@10, ..., LONG_NAM.@ZZ
```

This allows for modules whose names share the same first 8 characters.

Zero or more OBJ files will also be associated with a module. These are searched for using the module-path, and can be in different directories from their MOD files.

As an example, suppose we have a file C:\MODULES\DEF1.F90:

```
MODULE mod1
CONTAINS
   SUBROUTINE proc1
      PRINT *, 'HELLO'
   END SUBROUTINE
END MODULE
```

and the file C:\MODULES\DEF2.F90:

```
MODULE mod2
   USE mod1
CONTAINS
   SUBROUTINE proc2
      CALL proc1
   END SUBROUTINE
END MODULE
```

both of which are compiled using a command line of the form:

```
FTN95 DEF<n> /BINARY C:\OBJECTS\DEF<n>.OBJ
```

Now given a program C:\SOURCES\PROG1.F90:

```
PROGRAM test
   USE mod2
   CALL proc2
END
```

then to compile this program we would need either a) the environment variable MOD_PATH to include "C:\MODULES" or b) the command line:

```
C:\SOURCES>FTN95 PROG1 /MOD_PATH C:\MODULES
```

if no environment variable MOD_PATH had been set (or did not contain the path "C:\MODULES".)

To compile and link correctly we would need the command line:

```
C:\SOURCES>FTN95 PROG1 /MOD_PATH C:\MODULES;C:\OBJECTS /LINK
```

or the environment variable MOD_PATH to contain both "C:\MODULES" and "C:\OBJECTS".

If the program is compiled using SLINK or LINK77 directly, then the module's OBJ files will need to be specified in the list of OBJ files, e.g. with a DBOS compiler we would need to use the following sequence to link PROG1:

```
LOAD C:\SOURCES\PROG1.OBJ
LOAD C:\OBJECTS\DEF1.OBJ
LOAD C:\OBJECTS\DEF2.OBJ
FILE PROG1.EXE
```

11.

Fortran 95

The following article[1] describes the main differences between the Fortran 90 and the Fortran 95 standard and outlines the motivation for these changes.

FORALL

The FORALL statement and construct and PURE procedures have been added to Fortran 95 to allow Fortran 95 programs to execute efficiently in parallel on multi-processor systems. These features allow the majority of programs coded in High Performance Fortran (HPF) to run on a standard conforming Fortran 95 processor with little change. Adding these features to Fortran 95 does not imply that a particular Fortran 95 processor is multi-processor.

The purpose of the FORALL statement and construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement and the WHERE construct in Fortran 90. FORALL differs from these constructs in its syntax, which is intended to be more suggestive of local operations on each element of an array, and in its generality, which allows a larger class of array sections to be specified. In addition, a FORALL may invoke user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax).

FORALL is not intended to be a completely general parallel construct; for example, it does not express pipelined computations or multiple-instruction multiple-data (MIMD) computation well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics.

[1] written by Craig Dedo and used by permission.

Nested WHERE Construct

The WHERE construct has been extended in order to provide for syntactic regularity with the FORALL construct. The FORALL construct allows for nested FORALL constructs.

Early implementation of some High Performance Fortran (HPF) processors included the nested WHERE construct. Use of these processors showed that the feature provided real value to customers.

PURE procedures

The purpose of PURE procedures is to allow a processor to know when it is safe to implement a section of code as a parallel operation. The freedom from side effects of a pure function assists the efficient implementation of concurrent execution and allows the function to be invoked concurrently in a FORALL without such undesirable consequences as nondeterminism. It also forces some error checking on functions used in a FORALL construct.

Elemental procedures

ELEMENTAL procedures provide the programmer with more powerful expressive capabilities and the processor with additional opportunities for efficient parallelisation.

Extending the concept of elemental procedures from intrinsic to user-defined procedures is very much analogous to, but simpler than, extending the concept of generic procedures from intrinsic to user-defined procedures. Generic procedures were introduced for intrinsic procedures in Fortran 77 and extended to user-defined procedures in Fortran 90. Elemental procedures were introduced for intrinsic procedures in Fortran 90 and, because of their usefulness in parallel processing, it is quite natural that they be extended to user-defined procedures in Fortran 95. ELEMENTAL procedures are PURE,

Improved initialisations

A significant number of useful applications will be facilitated by the ability to perform more complicated calculations when specifying data objects. Allowing a restricted class of non-intrinsic functions in certain specification expressions achieves this goal.

Automatic deallocation

Automatic deallocation of allocatable arrays provides a more convenient and less error prone method of avoiding memory leaks by removing the burden of manual storage deallocation for allocatable arrays.

The “undefined” allocation status of Fortran 90 meant that an allocatable array could easily get into a state where it could not be further used in any way whatsoever. It could not be allocated, deallocated, referenced, or defined. The result was undefined if the array was used as the argument to the ALLOCATED function. Removal of this status provides the user with a safe way of handling unSAVEd allocatable arrays and permits use of the ALLOCATED intrinsic function to discover the current allocation status of the array at any time.

New initialisation features

The prime motivation for adding a means to specify default initialisation for objects of derived type is a desire to eliminate memory leakage, i.e., situations in which allocated memory becomes inaccessible. This can occur in applications which manipulate objects of derived type with pointer components.

Most memory leakage can be avoided if it is possible to specify that pointers be created with an initial status of disassociated.

This standard provides a new intrinsic function, NULL, with a single optional argument. NULL allows the initial status of a pointer to be specified as disassociated in declarations, structure constructors, or type definitions.

Several alternatives were considered and rejected for a variety of reasons. Most of these reasons involve the problem of disambiguating references to generic procedures; i.e., when a program invokes a generic procedure, which specific procedure is supposed to be invoked? Completely defining a pointer object before using it does not help with the disambiguation problem. Creating a NULL. constant does not provide any way to specify the type and type parameters of the pointer that is returned. A NULL function with a MOLD argument provides these needed capabilities.

This language extension does not completely solve the memory leakage problem; for that, an automatic destructor is needed which would be invoked for local pointers and structures with pointer components when the procedure in which they are created terminates. Such a facility is not included in this standard; it could be provided automatically by a processor that strove to conserve allocatable memory.

Automatic pointer destructors were not included in this standard because the consequences of inaccessible pointers are not as serious as they are with allocatable arrays which become inaccessible. If a local, unSAVEd pointer is not deallocated and the program exits the procedure in which it is defined, the storage is not recoverable but the pointer is reusable.

For reasons of determinacy and portability, an object for which default initialisation is specified is not allowed to appear in a DATA statement. In traditional implementations, a compiler passes initialisation information for non-dynamic variables to a loader, which is frequently designed to handle object code from several Different languages. It would be difficult to guarantee that initialisation in a DATA statement would override the default initialisation specified in a type definition.

Remove conflicts with IEC 559

Edits to sections 4.3.1 and 7.1.7 removed conflicts in Fortran 90 with the IEC 559 (IEEE 754/854) standard. Previous Fortran standards appeared to prohibit certain numeric values and numeric operations which were valid when using IEC 559 arithmetic. Fortran 90 requires that the value of a positive zero be the same as that of a negative zero, whereas IEC 559 requires that they be different. Fortran 90 prohibits any mathematically invalid operation, whereas IEC 559 requires such operations to produce +Inf , -Inf or NaNs.

These conflicts were removed to allow processors the ability to implement more of the IEC 559 arithmetic model without violating the Fortran standard.

Minimum width editing

Certain edit descriptors have been extended to allow formatted output of numeric values, while minimising the amount of "white space" produced. This extension permits more information to be presented on a limited width display device (e.g. terminal), without concern about overflowing a user specified field width. Allowing a zero field width to be specified for the I, B, 0, Z and F edit descriptors provides this functionality. A field width specification of zero always specifies a minimum field width; it never specifies suppression of data output or an actual field width of zero.

Namelist

Although not described here, namelist input has been extended to allow comments to follow one or more object name / value pairs on a line. This allows users to document how a namelist input file is structured, which values are valid, and the meaning of certain values. Each line in the namelist input may contain information about name / value pairs. This additional information may improve the documentation of the input file.

CPU-TIME intrinsic subroutine

A new intrinsic subroutine CPU-TIME has been added. This feature is provided to allow the assessment of which processor resources a piece of code consumes during execution. This could be the execution of the whole program or only a small part of it. Additional purposes could be comparing different algorithms on the same computer or trying to discover which parts of a calculation on a computer are most expensive.

MAXLOC and MINLOC intrinsics

Fortran 90 specified the MAXLOC and MINLOC intrinsics with only ARRAY and MASK arguments. The High Performance Fortran (HPF) group added the DIM argument between the original two arguments for consistency with the MAXVAL and MINVAL intrinsics. An incompatibility between Fortran 90 and HPF results unless MAXLOC and MINLOC are specified as generic interfaces, each with two specific interfaces: the first matching Fortran 90 and the second adding a non-optional DIM argument as the second argument. The Fortran Standards Committee decided that Fortran 95 should allow the DIM and MASK arguments to be specified in either order. For consistency, this provision for DIM and MASK to be specified in either order was extended to the other intrinsics which have the same arguments. These intrinsics are MAXVAL, MINVAL, PRODUCT, and SUM.

Deleted features[2]

The current list of deleted features is:

- real and double precision DO control variables,
- branching to an END IF from outside its block,
- PAUSE statement,
- ASSIGN, assigned GOTO, assigned FORMAT specifiers,
- cH edit descriptor,

New obsolescent features

There are some new obsolescent features to be added to those brought over from Fortran 90:

- computed GOTO,
- statement functions,
- DATA statements amongst executables,
- assumed-length character functions,
- fixed form source,
- assumed-size arrays (i.e. with '*' in the last dimension).
- CHARACTER*< *len* > style declarations.

Language Tidy-ups

Language tidy-ups are mainly concerned with ironing out ambiguities in the Fortran 90 standard and adding extra text to make matters clearer.

[2] Note that, with the exception of 'branching to ENDIF', features deleted from the Fortran standard are still available in FTN95. 'Branching to ENDIF' is not permitted in FTN95.

12.

Migrating to FTN95

Unsupported features

The following syntax forms which may occasionally appear in legacy code are not included in the Fortran 95 standard and are also not available in FTN95.

DATA I/%K/
This would initialise 'I' to the address of the statically allocated variable 'K'. Use I = LOC(K) to get the same effect.

Octal constants using ':177' notation
These could not be implemented because of ambiguities with array section syntax, e.g. is 'A(:177)' the 127th element of 'A', or is it a section from the start of the array to number 177? Use O'177' syntax instead.

Branching to END IF from outside the block
This is the only deleted feature not to be implemented. Alter the code to branch to the statement following the END IF.

VAX extensions
Only those VAX extensions that are now part of the Fortran 95 standard have been implemented in FTN95. Most of the VAX extensions that are not in this category have equivalent functionality in Fortran 95.

Deleted features

The following features have been deleted from the standard. With the exception of 'branching to ENDIF', these features are included in FTN95 but for portability should not be written into new code. Although these features will not be deleted from FTN95, in future versions, technical support in these areas will not be provided. Naturally these features are not available when the compiler option /ISO is used.

- REAL and DOUBLE PRECISION DO index variables

- Branching to END IF from outside the IF (see above)
- PAUSE statement
- ASSIGN and assigned GOTO
- Hollerith constants

Obsolescent features

The following features are marked as obsolescent in the standard. These features are included in FTN95 but should not be written into new code for compatibility with future Fortran versions. Although these features will not be deleted from FTN95, in future versions, technical support in these areas will not be provided.

- Arithmetic-IF
- Shared DO-loop termination, e.g.

```
      DO 10 I=1,10
      DO 20 K=1,10
        ...
   10 CONTINUE
```

- Alternate return.
- Computed GOTO.
- Statement functions.
- DATA statements among executable code.
- Assumed-length character functions.
- Fixed-format source code.
- assumed-size arrays (i.e. with '*' in the last dimension).
- CHARACTER*<len> declarations.

Converting INCLUDE files

It is recommended that FTN95 /CONVERT be used on any INCLUDE files to ensure these files are compatible with both fixed and free format source code.

13.

Language extensions

The list of extensions below is only available if the /ISO compiler option has not been specified. This is a subset of the FTN77 Fortran extensions because some FTN77 extensions are now part of the Fortran 95 standard (e.g. B, O and Z format descriptors).

- Use of Hollerith data anywhere in source (not just in FORMAT strings)
- Use of @ and $ characters in names and common block names
- Octal, hexadecimal and binary constants outside of DATA statements
- WHILE construct
- Internal procedures
- Conditional compilation
- Using an external alias in a sub program or INTERFACE block.
- In-line 32-bit assembler (see Chapter 14)
- Extra intrinsic functions (see Chapter 14)
- Byte alignment of data (see page 137)
- Input/output extensions (see Chapter 8)

Hollerith data

Hollerith data is not part of the Fortran standard although it was included in Fortran 66. In FTN95 they have been implemented as extensions to the Standard for compatibility with FTN77. 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(KIND=1) LOGICAL(KIND=1)	1
INTEGER(KIND=2) LOGICAL(KIND=2)	2
INTEGER(KIND=3) LOGICAL(KIND=3) REAL(KIND=1)	4
REAL(KIND=2)	8
REAL(KIND=3)	10

Hollerith data is allowed in FTN95 as follows:

1) In DATA statements for variables and arrays of type INTEGER and REAL. For example:

```
REAL(2) :: A(3)
INTEGER(3) :: IB(2)
DATA A/24HABCDEFGHIJKLMNOPQRSTUVWX/
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 *Aw* 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. FTN95 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(1) :: R
```

```
      LOGICAL(3) :: 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 the compile-time option /CHECK, 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. FTN95 allows the @ and $ 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 FTN95 library name or other reserved name. However, no system name contains a $ character.

Octal, hexadecimal and binary constants

FTN95 permits the use of B, O and Z constants anywhere in a program (the standard specifies these should only appear in DATA statements.) To specify the size of the constant, a kind specifier should appear after the constant, e.g. "Z'7F'_1" is an INTEGER(KIND=1) constant.

WHILE construct

For compatibility with FTN77, FTN95 offers an alternative WHILE construct. Its general form is

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

The WHILE-block may contain any executable Fortran statements. WHILE constructs can be nested within each other or within IF-, ELSE-, and ELSEIF-constructs and/or DO constructs. The rules of nesting are the same as those for the IF construct.

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
```

Internal procedures

FTN95 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. These are mainly provided for compatibility with FTN77, since Fortran 95 allows internal sub-programs via the CONTAINS statement, although internal procedures are still useful in run-time sensitive code.

As its name suggests, an internal procedure is local to the program unit in which it appears. FTN95 provides three statements to deal with internal procedures, and extends the use of the EXIT statement:

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).

Care should be taken to ensure that the name of the internal procedure is not the same as any construct-label in the routine otherwise the EXIT statement could be ambiguous.

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
```

Conditional compilation

FTN95 provides conditional compilation by means of the SPECIAL PARAMETER statement, the /SPARAM and /VPARAM compile-time options and the CIF, CELSE and CENDIF statements.

SPECIAL PARAMETER, /SPARAM and /VPARAM

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 or /VPARAM 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 when used with /SPARAM. The alternative is:

/VPARAM <name> <integer>

which allows individual values to be assigned by name rather than assigning one value to all. A value assigned by /VPARAM over-rides a value assigned by /SPARAM.

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> <op> <constant>) THEN
 . . .
CELSE
```

```
. . .
CENDIF
```

where <name> is a SPECIAL PARAMETER, <op> is one of the arithmetic relational operators (.EQ.,.LT., etc. or their equivalents ==, < etc.) and <constant> is an integer constant. In fixed format, 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 CELSE, CIF,.....CENDIF as in the Fortran block-IFstatement.

SPECIAL PARAMETERS can be assigned using /SPARAM, /VPARAM or by using a standard Fortran PARAMETER statement. Values that are not given a value, default to zero. If /SPARAM and /VPARAM are not used, then /FPP must be included in the compiler command line in order to invoke the preprocessor.

As a special case, the form

```
CIF (<name>) THEN
. . .
```

can be used where <name> is assigned to 1 (unity, representing TRUE) or to zero (representing FALSE).

Using an external alias

As an extension to the standard, the keyword ALIAS can be used in the definition of a FUNCTION or SUBROUTINE or in an INTERFACE block. The syntax is as follows:

SUBROUTINE <name> ([<args>]) [ALIAS <str>] [RESULT(<name>)]

with a similar form for a FUNCTION. If a subroutine takes no arguments, brackets for the empty argument list must be supplied. An ALIAS can be used when you need to export (to a non-Fortran environment) a routine with a name that includes lower case letters.

For example,

```
PROGRAM test
   INTERFACE
      SUBROUTINE foo(i) ALIAS 'TheFunction'
         INTEGER i
      END SUBROUTINE
   END INTERFACE
   CALL foo(99)
END

SUBROUTINE foo(i) ALIAS 'TheFunction'
   INTEGER i
   PRINT *, i
END
```

The SEQUENCE statement

The standard Fortran SEQUENCE statement is extended in FTN95 in order to allow an alignment size value. This is for interfacing with non-Fortran code. It has the following syntax:

SEQUENCE(<integer-constant>)

The <integer-constant> must be 1, 2, 4, 8 or 16 bytes and causes the elements of the TYPE to be aligned on these boundaries. The default alignment is on single byte boundaries. This extension does not provide a method for reducing the access time.

Note that the components of a TYPE that does not have the SEQUENCE attribute are re-ordered in order to improve alignment and cache utilisation.

14.

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. FTN95 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 the *FTN95 User's Supplement*.

The Win32 execution environment

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 unambiguously 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 non-local dynamic 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.

Labels

It is possible to jump from Assembler to labelled Fortran statements and vice-versa. 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%.
- Non-local dynamic 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 FOO
CODE
  MOVB     FOO+3,=32       ;Sets FOO(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 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:

```
MOVS                ;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.

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 FTN95. The following pseudo instructions have been provided:

- In-line data may be inserted using DB, DW, or DD pseudo instructions. For example:

```
DD    Z'FEFFFFFF'       ;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
```

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, DCORE8, XCORE10, ZCORE8, ZCORE16, and ZCORE20 are available to examine the contents of a given location.

  Each function takes an INTEGER(KIND=3) argument which is an address and returns the following value at that address.

CCORE1	CHARACTER
CORE1	INTEGER(KIND=1)
CORE2	INTEGER(KIND=2)
CORE4	INTEGER(KIND=3)
FCORE4	REAL(KIND=1)
DCORE8	REAL(KIND=2)
XCORE10	REAL(KIND=3)
ZCORE8	COMPLEX(KIND=1)
ZCORE16	COMPLEX(KIND=2)
ZCORE 20	COMPLEX(KIND=3)

  These functions 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.

- 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.

Support for MMX or SSE extensions

Some support is included to aid users wishing to use MMX or SSE extensions.

The intrinsic function CPUID@ takes two arguments, the first being the input value, the second being any data type that is larger than 16 bytes (usually a rank one INTEGER(3) array of size 4). This function returns the EAX, EBX, ECX and EDX values returned from a “cpuid” instruction. Two other intrinsics (that use CPUID@) are: HAS_MMX@ and HAS_SSE@. These are both LOGICAL(3) functions with no arguments that return .TRUE. if the feature is available on the host machine. CODE..EDOC fully supports the new MMX and SSE instructions.

The type attribute ALIGN(n) forces a variable to be aligned to a different boundary than is usual for the data type. n is 8, 16, 32, 64 or 128 and is the size (in bits) of the alignment required. For example:

```
INTEGER(2), ALIGN(32) :: a, b, c
```

would declare three two-byte integers that are aligned on four-byte boundaries. This extension to standard Fortran 95 should be used with care as incorrect alignment can result in code running slower or not at all (if it uses some SSE instructions.)

15.

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. It is important to note that you can freely mix FTN95 and FTN77 compiled code and either or both can do the input/output.

Data types

Basic data types

The table 15-1 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)`

FTN95 `INTEGER (KIND=3) numbers(10,20)`

C/C++ `int numbers[20][10];`

Data type	Size (bytes)	C/C++	FTN77	FTN95
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	-	REAL (KIND=3)
Character	1	char	CHARACTER	CHARACTER

Table 15-1

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]`. In Fortran by default the elements are actually numbered from one whilst in C/C++ they are numbered from zero.

Character strings

C/C++ character strings are stored as a null (char(0)) terminated arrays of characters whilst standard Fortran characters 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 FTN95 from C/C++

Introduction

The following text assumes that you are writing in Salford C/C++ and are calling an FTN95 relocatable binary library (RLB) or dynamic link library (DLL).

When calling FTN95 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

Standard Fortran character arguments are fixed length and padded with space characters. In order to determine the length of a character argument, the FTN95 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 (LEN=*)::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. Is is therefore necessary to provide an interface routine between the FTN95 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 FTN95 variable (see table 15-1). 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++.

Calling C/C++ from FTN95

The C_EXTERNAL and STDCALL keywords give the Fortran programmer the ability to call C/C++ routines, forcing the compiler to generate extra code to handle some of the data conversions. C_EXTERNAL is used when the "cdecl" calling convention is used by C/C++ (e.g. the Win16 Windows API). STDCALL is used when the "stdcall" calling convention is used by C/C++ (e.g. the Win32 Windows API).

An example of a type requiring data conversion is the string data type. As we have already noted, C/C++ strings are null terminated, whilst standard Fortran strings are fixed length, padded with spaces.

As an extension to the Fortran standard, a string literal that is followed by the letter C (e.g. "string"C) is interpreted as a null terminated string and can be used as an argument in a call to a C/C++ function. In certain simple cases, the use of this extension makes it unnecessary to include the C_EXTERNAL keyword (by default, the order of the arguments is given by C_EXTERNAL rather than STDCALL, so if

other conversions or extensions like call-by-value are not needed, C_EXTERNAL becomes redundant).

The C_EXTERNAL and STDCALL declarations for a function inform 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. Code is also generated after the function call 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 (for STDCALL, simply replace C_EXTERNAL by STDCALL or refer to the *ClearWin+ User's Guide*).

```
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). This appears in single quotes and is case-sensitive.

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(KIND=3)::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(KIND=3)::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 and 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(LEN=*) 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

FTN95 contains two further keywords, WINREF and WINSTRING. These are only used with C_EXTERNAL and 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.

Calling an FTN95 DLL from Visual Basic

An FTN95 DLL that is called by an executable that uses the STDCALL calling convention (e.g. one created using Win32 Visual Basic) must use F_STDCALL in the declaration of all exported subprograms. For example,

```
F_STDCALL FUNCTION F(X)
INTEGER F,X
```

```
F=X
END
```

Such a function could make calls to Windows API functions provided an interface is provided via, for example, "USE MSWIN" or "INCLUDE <windows.ins>". Alternatively an interface for each API function can be given explicitly. For example,

```
STDCALL SENDMESSAGE 'SendMessageA' (VAL,VAL,VAL,VAL):INTEGER*4
```

A call is automatically made to a DLL function called LIBMAIN when the DLL is loaded. If you do not provide a definition of LIBMAIN then SLINK will provide a default for you. LIBMAIN is used to initialise global data. It takes the following form.

```
F_STDCALL INTEGER FUNCTION LIBMAIN(hInst,ul,lpR)
INTEGER hInst,ul,lpR
!***** Initialise global data here
LIBMAIN=1
END
```

A simple SLINK script would take the form:

```
dll
lo f95.obj
exportall
file c:\windows\f95.dll
```

A Visual Basic program can use calls to the API functions **LoadLibrary** and **FreeLibrary** in order to ensure that the DLL does not unload whilst a Visual Basic form is loaded. Here is some sample code for this purpose.

```
Private Declare Function LoadLibrary Lib "kernel32" Alias
"LoadLibraryA" (ByVal s As String) As Long
Private Declare Function FreeLibrary Lib "kernel32" (h As Long)
As Long
Dim hLibModule As Long

Private Sub Form_Load()
hLibModule = LoadLibrary("f95.dll")
End Sub

Private Sub Form_Unload(Cancel As Integer)
i& = FreeLibrary(hLibModule)
End Sub
```

The FTN95 DLL is assumed to reside in the Windows system directory.

Mixing I/O systems in Fortran and C/C++

You can freely mix FTN95 and FTN77 compiled code and either or both can do the input/output.

However, in general the I/O systems in Fortran and C/C++ are different and should not be mixed. For example, it is not usually possible to open a file in FTN95, 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.

16.

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 15), 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 that one occurance 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 153).

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 compulsary 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 16-1.

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 (KIND=1), CHARACTER	char
INTEGER (KIND=2), LOGICAL (KIND=2)	short int
INTEGER (KIND=3), LOGICAL (KIND=3)	int
REAL (KIND=1)	float
REAL (KIND=2)	double
CHARACTER*(*x*)	char [*x*]

Table 16-1

Limitations

The following limitations apply to the source file.

Maxmum line length	160 characters
Maxmum number of names	5000
Maxmum name length	40
Maxmum nesting level for include files	10
Maxmum number of common blocks	29

17.

SLINK

For information about the Salford DOS/Win16 linker LINK77 see the *FTN95 User's Supplement*.

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 Executables (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 preceeded 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 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:

`lo object1.obj`	creates references to KERNEL32.DLL
`lo kernel32.dll`	satisfies current references to KERNEL32.DLL
`lo 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

A list of DLLs included under the heading "system DLLs" is given in the reference section beginning on page 167.

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.

An FTN95 DLL that is called by an executable that uses the STDCALL calling convention (e.g. one created using Win32 Visual Basic) must use F_STDCALL in the declaration of all exported subprograms. For example,

```
F_STDCALL FUNCTION F(X)
INTEGER F,X
F=X
END
```

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 on page 167 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. You can use a wild card form for *filename* (e.g. *.obj).

Alternatively, a list of COFF objects to be included may be inserted in a ***listfile*** the path of which is preceeded by an **@**.

archive (implib) *filename*

Specifies that an archive is to be generated from objects loaded with the **addobj** command. It also specifes 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 delimitted 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]] [data|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. Overides the **exportx** command.

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 specifed 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

hardfail

Causes the link to be abandoned if there are unresolved externals after all objects and libraries have been scanned.

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

maximimum 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
```

insert *filename*
Allows a script file to be inserted into another script file.

listunresolved (lure)
Prints a list of external references which are currently 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. A list is automatically presented when all objects have been scanned (unless **nolistunresolved** is used).

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 is not 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.

logfile *filename*
Directs console output to *filename*.

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.

notify [*symbol*]
This causes the occurrence of ***symbol*** in an object file or COFF member to be reported (e.g. notify _GetVersion@0). Use **notify** without ***symbol*** to disable reporting.

notrace
Suppresses the generation of the internal map within the executable. Without this map a runtime traceback is impossible.

nolistunresolved (nolure)
Prevents the listing of unresolved references when all objects have been scanned. However, unresolved common references will still be reported and will terminate a link session.

no_external_data_search (neds)
The **no_external_data_search** (**neds**) and **external_data_search** (**eds**) commands disable and re-enable explicit searching for data initialisations in libraries. The default is **external_data_search** (i.e. libraries are searched for data

initialisations). External data searching can be disabled and re-enabled at any stage during an interactive mode linking session or globally during a command line mode link.

permit_duplicates (pd)
Permits multiple definitions of external symbols. The first definition is used.

quit (q)
Immediately exits SLINK. No output files are produced.

relax
relax and **strict** are used to disable and re-enable checking of code to data matching. The default is **strict** (i.e. checking is enabled).

report_debug_files [on | off]
Reports files that contain debug information. If the parameter is omitted, the default is **on**. In any case SLINK will automatically link for debug if it finds such files.

report_scan_process (rsp)
This command is designed to assist in the development of static libraries. The command prints out a diagnostic trace of the link in progress, showing which files and COFF members are being scanned. The external references that are being created and the symbol that has triggered the load of an archive member are also printed (together with the file and archive member it came from).

rlo
Normally, functions imported from DLLs are linked at run time by the system loader. Functions that are not present in the DLL will not be noted as missing unless and until they are called. When **rlo** is used with SLINK, loading is aborted at run time if there are any missing functions.

shortnames
This is used when building an archive. Member names are truncated down to just the file name portion.

silent
Suppresses linker messages in interactive mode. Messages that confirm the creation of the executable etc. and error messages are unaffected.

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 maximimum size of ***reserve***. The ***reserve*** and ***commit*** values are rounded to 4 byte boundaries.

defaults,

for Salford programs:
reserve = 0x3200000 (50Mb)

commit = 0x4000 (16Kb)

for other programs:
reserve = 0x100000 (1Mb)

commit = 0x1000 (4Kb)

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 must be one of the following:

native no subsystem required

windows [,*major*[.*minor*]] a graphical user interface subsystem is required (***major*** and ***minor*** are optional version numbers).

console application requires only a character mode subsytem (but using a GUI is not precluded).

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

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:{*@listfile* | *filename*}

-align:#

-archive:[*filename*] alias for **-implib**

-base:***address***

-debug [:full | :partial | :none]

-decorate

-dll [:*modulename*] alias for **-library**

-entry:***symbol***

-export:***entryname*=[*internalname*] [,@ordinal[,noname]][,data|,constant]**

-exportall

-exportx:***entryname***

-file [:*filename*]	alias for **-out**
-filealign:#	
-hardfail	
-heap:*reserve* [,*commit*]	
-help	alias for **-?**
-imagealign:#	alias for **-align**
-imagebase:***address***	alias for **-base**
-implib [:*filename*]	
-library [:*filename*]	
-logfile:***filename***	
-map [:*filename*]	
-no_internal_map	alias for **-notrace**
-nolistunresolved	
-notify [:*symbol*]	
-notrace	
- no_external_data_search	
-out [:*filename*]	
-permit_duplicates	
-report_debug_files	
-shortnames	
-stack:***reserve*[,*commit*]**	
-subsystem:{native | windows | console}	
-vc[:*baseaddress*[,*commit*]]	alias for **-virtualcommon**
-virtualcommon[:*baseaddress*[,*commit*]]	

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.

18.

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.f90*, contains only a main program; the second *sub.f90* contains all of the user-defined subroutines that are called from the main progam. 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.f90
          ftn95 prog /check

sub.obj:  sub.f90
          ftn95 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.f90* and *prog.obj* is created by calling FTN95 using *prog.f90* with the /CHECK option. A similar dependency relationship and rule is used for *sub.obj*.

If *sub.f90* only were changed (for example) then calling the utility would result in *sub.f90* being recompiled (but not *prog.f90*). 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: .f90 .obj
```

```
.f90.obj:
          ftn95 $< /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 ".f90" 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 ".f90" file and that the object file is formed by calling FTN95 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: .f90 .obj

.f90.obj:
          ftn95 $< /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: .f90 .obj .lnk .exe

.f90.obj:
          ftn95 $< /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: .f90 .obj

OBJFILES=prog.obj \
         sub.obj
T=prog

.f90.obj:
         ftn95 $< /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: .f90 .obj .lnk .exe

.f90.obj:
          ftn95 $< /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: .f90 .obj .exe

OPTIONS=

OBJFILES=
```

```
.f90.obj:
        ftn95 $< $(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 *filename*	Use *filename* instead of the default file called *makefile*. A minus sign in place of *filename* 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 *default.mk*.
-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 .IGNORE:.
-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 *default.mk*.
-s	Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.
-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.

19.

Using AUTOMAKE

Introduction

AUTOMAKE is a utility from Polyhedron Software Ltd. It is part of the Polyhedron *plus*FORT suite and is released under licence by special permission. Copyright is reserved by Polyhedron. This chapter is taken directly from the Polyhedron documentation.

What does it do?

AUTOMAKE is a simple-to-use tool for re-building object and executable code after you have made changes to the Fortran and/or C source code. It examines the creation times of all the source and object code files, and recompiles wherever it finds that an object file is non-existent, empty or out of date. In doing this, it takes account not only of changes or additions to the source code files, but also changes or additions to INCLUDEd files - even nested INCLUDE files. For example, if you change a file which is INCLUDEd in half a dozen source files, AUTOMAKE ensures that these files are re-compiled.

How does it do that?

AUTOMAKE stores details of the dependencies in your program (e.g. file A INCLUDEs file B) in a dependency file, usually called *automake.dep*. AUTOMAKE uses this data to deduce which files need to be compiled when you make a change. Unlike conventional MAKE utilities, which require the user to specify dependencies explicitly, AUTOMAKE creates and maintains this data itself. To do this, AUTOMAKE periodically scans source files to look for INCLUDE statements. This is a very fast process, which adds very little to the overall time taken to complete the update.

How do I set it up?

The operation of AUTOMAKE is controlled by a configuration file which contains the default compiler name and options, INCLUDE file search rule etc.. For simple situations, where the source code to be compiled is in a single directory, and builds into a single executable, it will probably be possible to use the system default configuration file. In that case there is no need for any customisation of AUTOMAKE - just type AM (*am.bat* is the name of a Polyhedron batch file) to update both your program and the dependency file.

In other cases, you may wish to change the default compiler name or options, add a special link command, or change the INCLUDE file search rule; this can be achieved by customising a local copy of the AUTOMAKE configuration file. More complex systems, perhaps involving multiple programs, or source code spread across several directories, can also be handled in this way.

What can go wrong?

Not much. AUTOMAKE is very forgiving. For example, you can mix manual and AUTOMAKE controlled updates without any ill effects. You can even delete the dependency file without causing more than a pause while AUTOMAKE regenerates the dependency data. In fact, this is the recommended procedure if you do manage to get into a knot.

Running AUTOMAKE

To run AUTOMAKE, simply type AM. This invokes a batch file from the installation directory. The operation of AUTOMAKE is controlled by a configuration file, which is described in the next section. There is seldom any reason to modify the command file, though it is very simple to do so if required. It consists of two (or three) operations:

- Execute AUTOMAKE. AUTOMAKE determines what needs to be done in order to update the system, and writes a command file to do it.

The switches which may be appended to the AUTOMAKE command are:

TO= specifies the name of the output command file created by AUTOMAKE (the default is *amtemp.bat*).

FIG= specifies the name of the AUTOMAKE configuration file (the default is *automake.fig*). If no path is specified, AUTOMAKE follows the search procedure described in section 1.3 of the *plus*FORT manual.

- Execute the command file created by AUTOMAKE.

□□Delete the command file created by AUTOMAKE. This step is, of course, optional.

The AUTOMAKE Configuration File

The AUTOMAKE configuration file is used to specify the name of your compile and link procedures, and other details required by AUTOMAKE. One or more sample configuration files is available in the installation directory, and we suggest that you use one of these if possible, and make changes only if required to meet your requirements. The configuration file consists of a series of records of the form

```
keyword=value
```

or

```
keyword
```

where 'keyword' is an alphanumeric keyword name, and 'value' is the string of characters assigned to the keyword. The keyword name must begin in column 1. Any record with a space in column 1 is regarded as a comment. The keywords which may be inserted in the configuration file are:

COMPILE= specifies the command to be used to invoke the compiler. The command may contain place markers, which are expanded as necessary before the command is executed. For simple use, all you need to know is that the string '%SD%SF%SE' expands to the full name of the file to be compiled.

Place markers are discussed in the next section. The place markers that can be used within the compile command are %SD (name of directory containing source code), %SF (source file name), %SE (source file extension), %OD (name of directory containing object code), %OE (object file extension) and %ID (INCLUDE file search list).

```
COMPILE=ftn95 /check %SD%SF
```

It is possible to invoke the compiler using a batch file. However, it is necessary to preface the batch file name with 'CALL' or 'COMMAND/C'. For example

```
COMPILE=CALL fcomp %SD%SF%SE
```

FILES= specifies the names of files which are candidates for re-compilation. The value field should contain a single filename optionally including wild-cards. e.g.

```
FILES=*.for
```

INCLUDE= may be used to specify the INCLUDE file search list. If no path is specified for an INCLUDEd file, AUTOMAKE looks first in the directory which contains the source file, and after that, in the directories specified using this keyword. The directory names must be separated by semi-colons. For example, we might have:

```
INCLUDE=C:\include;C:\include\sys
```

Note that the compiler will also have to be told where to look for INCLUDEd files.

SYSINCLUDE= may be used to specify the search list for system INCLUDE files (i.e. any enclosed in angled brackets), as in

```
#include <stat.h>
```

If no path is specified, AUTOMAKE looks in the directories specified using this keyword. It does not look in the current directory for system INCLUDE files unless explicitly instructed to. The directory names following SYSINCLUDE= must be separated by semi-colons.

OBJDIR= may be used to specify the name of the directory in which AUTOMAKE is to look for object files. e.g.

```
OBJDIR=OBJ\
```

If OBJDIR= is not specified, AUTOMAKE assumes that source and object files are in the same directory. Note that if source and object files are not in the same directory, the compiler will also have to be told where to put object files.

OBJEXT= may be used to specify a non-standard object file extension. For example to specify that object files have the extension '.abc', specify

```
OBJEXT=ABC
```

This option may be useful for dealing with unusual compilers, but more commonly to allow AUTOMAKE to deal with processes other than compilation (for example, you could use AUTOMAKE to ensure that all altered source files are run through a preprocessor prior to compilation).

LINK= specifies a command which may be used to update the program or library file once the object code is up to date. For example, you could invoke a batch file called *l.bat* by specifying

```
LINK=CALL L
```

Place markers are allowed in the command specified using LINK=. The %OD (name of directory containing object code), %OE (object file extension) and %EX (executable file name) place markers are allowed in this context. See page 195 for a description of place markers. If there is no LINK= keyword, AUTOMAKE will update the program object code, but will not attempt to re-link.

TARGET= specifies the name of the program or library file which is to be built from the object code. Note that you will also have to tell the linker the name of the target file. The TARGET= keyword is required only if the LINK= keyword is specified.

DEP= may be used to over-ride the default dependency file name. For example:

```
DEP=THISPROG.DEP
```

causes AUTOMAKE to store dependency data in *thisprog.dep* instead of *automake.dep*.

SALFORDOBJ If this keyword is specified, AUTOMAKE checks for object files which are the result of a failed compilation using a Salford Software compiler. If it is not specified, AUTOMAKE may assume the object code is up-to-date, even if some compilations failed.

CHECK= may be used to specify a command to be inserted after each compilation. A typical application would be to check for compilation errors. For example,

```
CHECK=IF ERRORLEVEL 2 GOTO QUIT
```

would cause the update procedure to abort if there is a compilation error.

DEBUG Causes AUTOMAKE to list the contents of the dependency file to the screen before and after it is updated.

Place Markers

Before a compile or link command is executed, any place markers within the command are replaced by the string to which they refer.

%SD expands to the directory part of the file name which was specified using the

FILES= keyword (see page 193).

%SF expands to the name part (excluding the directory and extension) of the file to be compiled.

%SE expands to the extension part of the file name which was specified using the FILES= keyword. For example, if

```
files=e:\source\a*.for
```

was specified, and AUTOMAKE determines that *e:\source\aprog.for* must be compiled, then %SD expands to 'e:\source\' (including the final '\'), %SF to 'aprog' and %SE to '.for' (including the '.').

%OD expands to the name of the directory containing the object files (as specified using the OBJDIR= keyword - see page 194).

%OE expands to the extension used for object files normally '.OBJ', but can be modified using the OBJEXT= keyword - see page 194).

%ID expands to the include directory search path (as specified using the INCLUDE= keyword - see page 194).

%EX expands to the name of the target executable file (as specified using the TARGET= keyword - see page 195).

Place markers represented by 2, 3 or 4 adjacent '=' characters, as used in version 3.xx of *plus*FORT are no longer supported in version 5.

Multiple Phase Compilation

Sometimes, more than one compilation phase is required. For example, if source files are stored in more than one directory, you will need a separate compilation phase for each directory. Multiple phases are also required if you have mixed C and Fortran source, or if you need special compilation options for particular source files.

The 'GO' keyword may be inserted in your configuration file to add a new compilation phase. When AUTOMAKE reads a 'GO' keyword, it issues commands to update the object code, based on the current settings of FILES=, COMPILE=, INCLUDE=, OBJDIR= and OBJEXT=. It then returns to read further instructions which may change one or more of these keywords, and in the next phase, issues further commands based on the revised settings.

The number of phases is always one more than the number of 'GO' keywords in the configuration file.

Notes

□□As AUTOMAKE executes, it issues brief messages to explain the reasons for all compilations. It also indicates when it is scanning through a file to look for INCLUDE statements. Often AUTOMAKE will not find it necessary to scan a file until some time after you have changed it, so don't worry if these messages relate to files you haven't changed recently (the rule is that AUTOMAKE scans a file if the object code is more recent than the source code, but the source code has been changed since the last scan).

□□If for any reason the dependency file is deleted, AUTOMAKE will create a new one. Execution of the first AUTOMAKE will be slower than usual, because of the need to regenerate the dependency data.

□□AUTOMAKE recognises the INCLUDE statements in all common variants of Fortran and C, and can be used with both languages.

□□When AUTOMAKE scans source code to see if it contains INCLUDE statements, it recognises the following generalised format:

(a) Optional spaces at the beginning of the line followed by..

(b) An optional compiler control character, '%', '$' or '#', followed by..

(c) The word INCLUDE (case insensitive) followed by..

(d) An optional colon followed by..

(e) The file name, optionally enclosed between apostrophes, quotes or angled brackets. If the file name is enclosed in angled brackets, it is assumed to be in one of the directories specified using the SYSINCLUDE keyword (see page 194). Otherwise, AUTOMAKE looks in the source file directory, and if it is not there, in the directories specified using the INCLUDE keyword (see page 194).

□□If AUTOMAKE cannot find an INCLUDE file, it reports the fact to the screen and ignores the dependency relationship.

20.

Using Plato

Introduction

Plato is a Win32 editor that uses a MDI (Multiple Document Interface) and allows unlimited undo, keyword help, colour syntax highlighting and supports all of Salford's Win32 compilers.

This chapter describes how to use the Plato Integrated Development Environment, and shows how it is possible to compile, link and execute Salford programs from within the IDE.

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 your compilers and help files. To do this select **ShowProducts** 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's 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.

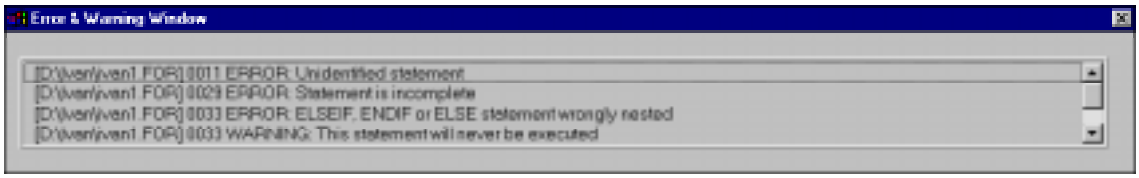
When a file is opened, the name is recorded in the Most Recently Used (MRU) list on the file menu, this list is saved and restored the next time Plato is started. The file name is also placed into a drop down list box on the toolbar.

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 move to the line where an error occurred by double clicking on the appropriate line in the Message window.

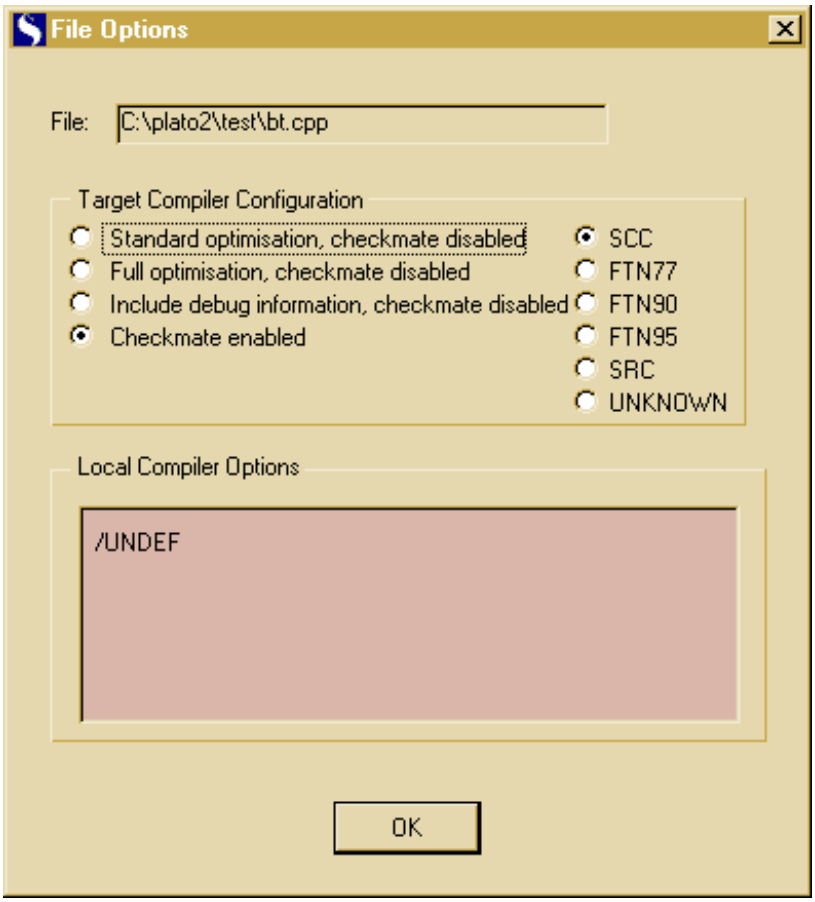


4) Once you are happy with your compilation, choose build  which will link your program.

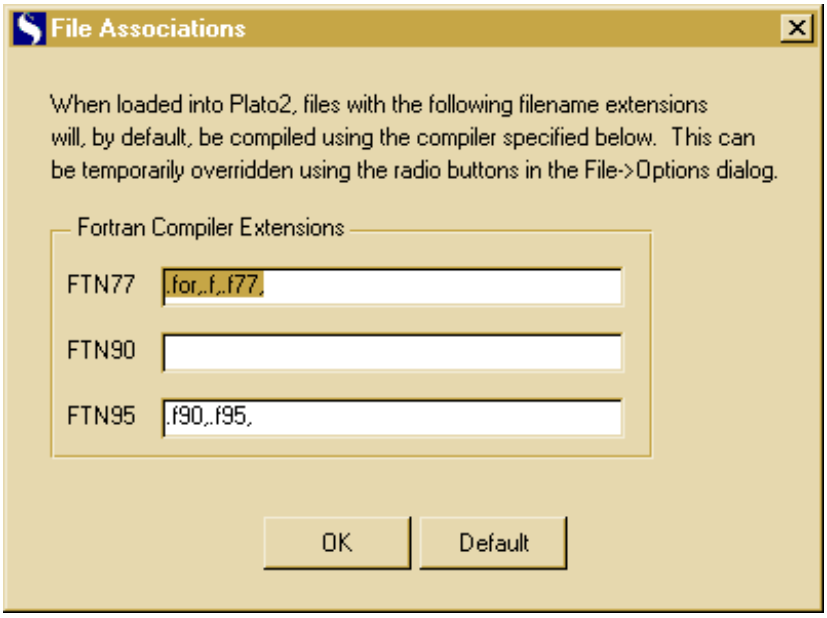
5) If the linking is successful choose **Run** (or click **OK** and then the  toolbar button). A window appears showing you the file to be executed and two radio buttons. If you select **Console**, Plato will open up a console before running the file.

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.



You can use the above dialog to change the compiler that will be used with the specified file. However, if you want Plato to always remember that files with a certain extension are to be associated with a default compiler, use **File Associations** from the **File** menu item.



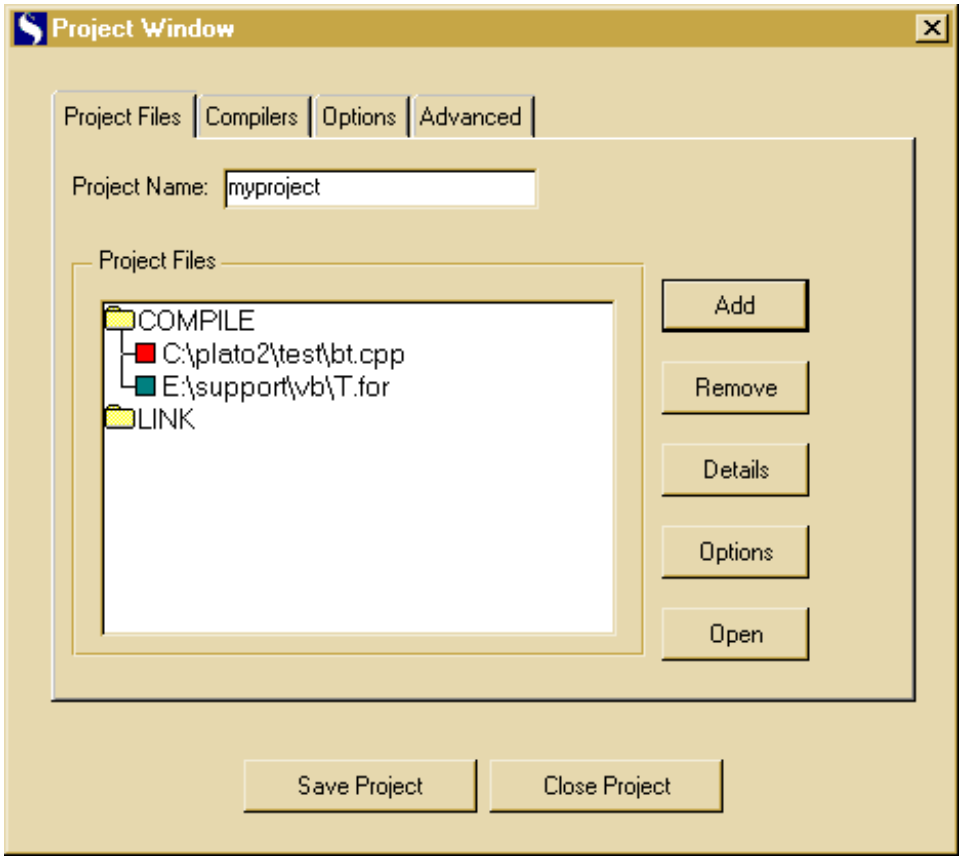
Working with projects

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 project name.
3) Build a list of source files. Left mouse click on the **Compile** folder in the Application tree and select the **Add** button.
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.



5) To specify compiler options for a particular file click the left mouse button on the filename and choose the **Options** button. To supply compiler options that will affect all files use the Global Compiler Options edit box in the Options property sheet.

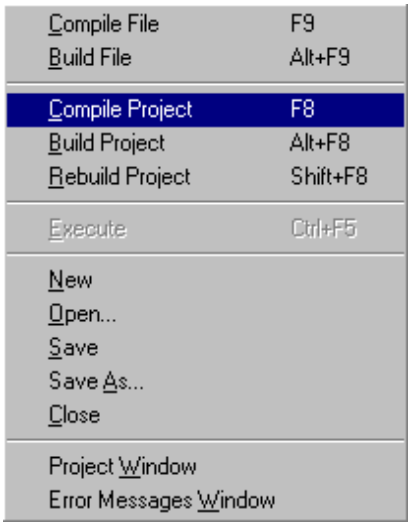
6) Save the project with the **Save Project** button.

7) 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



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.

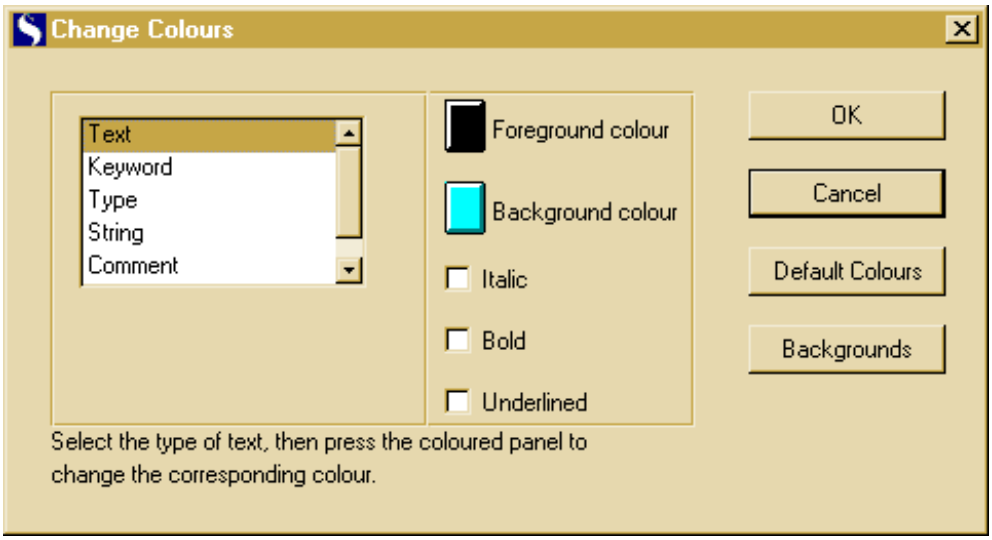
Projects - Advanced Features

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 of the Compilers property sheet 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 on the Advanced property sheet.

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 **Set 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

21.

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. 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 95 will, in all probability, differ from those chosen in any other implementation for the same error conditions.
- IOSTAT values of 10000 and above indicate that a file system error has occurred. The system error code is determined by subtracting 10000. System documentation can then be used to decode the error. Alternatively, the error message can be obtained by passing the unadjusted IOSTAT value to the Salford library routine FORTRAN_ERROR_MESSAGE@. When an IOSTAT value is returned that corresponds to a file system error (i.e. a value greater than 10000), the system error code can also be obtained by calling the Salford library function GET_LAST_IO_ERROR@.

Error No.	Message
0	No error
1	Floating point arithmetic overflow
2	Integer arithmetic overflow
3	Argument to CHAR outside range 0 - 255

4	Character argument/function name of wrong length
5	Attempt to execute invalid assigned GOTO
6	Inconsistent call to routine
7	DO-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*w.m* run-time format
55	*m* > *w* in O*w.m*
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
192	Substring required, not an array index
193	Pascal run time error
194	Reference through NULL pointer
195	Reference through invalid pointer
196	Expected ON or OFF
197	Requested a negative amount of memory
198	Not enough storage heap to open window
199	Run out of heap space
200	Buffer overflow in namelist-directed I/O
201	End-of-file in namelist input group
202	Syntax error encountered while reading namelist group
203	Too many values supplied for input array in namelist-directed input/output

204	Invalid identifier in namelist input group
205	Incorrect number of subscripts in namelist input
206	Subscript or substring out-of-range in namelist input
207	Invalid SHARE specifier
208	You may not read from a file that is write-only
209	Invalid ACTION specifier
210	Invalid POSITION specifier
211	Invalid DELIM specifier
212	Invalid PAD specifier
213	Invalid ADVANCE specifier
214	You cannot use an ADVANCE specifier here
215	Input record too short
221	+, - or digit expected
222	1st parameter is undefined
223	1st string is undefined
224	2nd parameter is undefined
225	2nd string is null
226	2nd string is undefined
227	Assign not allowed on an open file
228	Bad file control block
229	DA access not allowed for text files
230	DA access not allowed to terminal or string
231	DA not allowed for text files
232	Digit expected
233	DOS pathname unobtainable
234	End of file
235	EOLN used at EOF
236	Failed to position to end of file

237	Failed to position to record
238	File cannot be closed
239	File cannot be emptied
240	File cannot be opened
241	File cannot be positioned
242	File cannot be reopened
243	File cannot be rewound
244	File has not been opened
245	File is not open for reading
246	File is not open for writing
247	File is open for reading
248	File is open for writing
249	File is read-only
250	File OUTPUT may not be reset
251	File size is not a multiple of record size
252	Heap slot has already been disposed
253	Heap slot has been corrupted
254	Heap slot is of the wrong size
255	Invalid record number
256	No EOLN before EOF
257	No storage available
258	Not enough room for insertion
259	Not enough room for result
260	Number too large
261	Pascal system error
262	Pointer is NIL
263	Pointer is undefined
264	Range error

265	Set range error
266	String files not allowed
267	String is null
268	String is undefined
269	This operation only allowed for DA files
270	Too many temporary files
271	Undefined Boolean
272	while reading from file
273	while writing to file
274	Reference through invalid C pointer
275	Reference through NULL pointer %F
276	Reference through un-initialised pointer %F
277	Attempt to overwrite constant data
278	Corrupt storage availability index
279	Attempt to call Windows application routine when not in windows
280	Run out of Windows selectors
281	String is not terminated %F
282	Invalid long jump %F
283	Search has spilled outside storage area %F
284	Negative count not permitted %F
285	Concatenation would spill outside storage area %F
286	Read has spilled outside storage area %F
287	Conversion has spilled outside storage area
288	Reference through dangling C pointer %F
289	Reference to a portion of the format buffer which has been overwritten %F
290	Essential arguments missing %F
291	Not enough arguments to satisfy format %F
292	Pointer does not point at a function

293	Pointer references code - not data %F
294	Storage block is not big enough for this operation %F
295	Impossible number of bytes in copy operation %F
296	Invalid FILE structure
297	Invalid double passed to printf
298	Invalid long double passed to printf
299	Reference to missing C argument
300	Corrupt destructor chain
301	Attempt to create invalid SAI entry
302	Attempt to 'free' memory allocated by 'new' - use 'delete'
303	Attempt to 'delete' memory not allocated by 'new'
304	This operation is not allowed on a windows stream
305	Input/Output failure in the C library.
306	Read operation has failed
307	Write operation has failed
308	Illegal attempt to close standard streams (stdin, stdout or stderr)
309	Cannot read from a stream after writing without an intervening seek or fflush
310	Cannot write to a stream after reading without an intervening seek.
311	Unable to find length information to delete whole array with destructors
312	Attempt to call a pure virtual function implicitly
313	Illegal attempt to write to a read only file.
314	Illegal attempt to read from a write only file.
315	Illegal attempt to use low level IO functions on a file opened by ANSI C IO functions
316	Object contains corrupt virtual function table pointer
317	Pointer to member function has not been defined
318	C library floating point error handler unexpectedly called.
319	Attempt to exit a function without a return value

320	Attempt to create overlapping SAI entry
321	Invalid size must be > 0
322	#define CREATE_CONSTRUCTOR
323	#define SEEKMODE_SEEKOFFSET
324	Invalid file handle
325	Null buffer pointer or zero buffer size
326	Automatic allocation of buffer failure
327	Attempt to use an output stream for input - invalid mode
328	Fail to open file
329	Invalid index
330	Available space is full
331	#define STDIOSTRM_CONSTRUCTOR
332	Invalid file mode - ios::in, ios::out or ios::app
333	Invalid seek_mode - ios::beg, ios::end or ios::cur
334	Attempt to write beyond buffer supplied to istream object
335	#define CREATE_COUT
336	#define CREATE_CIN
337	Object already buffered or invalid buffer pointer or size
338	#define STRSTREAMBUF_ALLOCATION
339	Null pointer supplied to stream function
340	Object is not a streambuf class
341	Invalid window handle
342	Reference to position beyond buffer
343	Invalid stream pointer (FILE*=NULL)
358	Special failed number type 2
359	Special failed number type 1
360	Invalid disk drive number (0=A, 1=B, etc..)
361	Invalid file mode

362	Invalid argument
363	Invalid Clearwin window's handle
364	Invalid dialog handle
370	get_real_mode_memory_region can only be used on real mode memory (i.e. 0 to 0x10000000)
371	Windows program could not be started (is LAUNCHER running?)
372	Attempt to start Windows program outside Windows
373	No characters read by scanf
374	This operation requires a Pentium-class processor
375	Attempt to free part of a storage block
376	Attempt to delete unallocated storage
377	Attempt to free unallocated storage
378	Coprocessor stack is not empty (CHECKSTACK@)
379	Attempting to close a standard stream is illegal
380	Zero or negative array size not permitted
381	FTN90 run-time error
382	Coprocessor stack overflow
383	Coprocessor stack underflow
384	Second argument to NEAREST is zero
385	Real and imaginary parts of the argument to complex LOG are both zero
386	The FROMPOS, LEN and TOPOS arguments of MVBITS must be nonnegative and not exceed BITSIZE(FROM)
387	FROMPOS+LEN and TOPOS+LEN must not exceed BITSIZE(FROM)
388	DIM out of range
389	Nonconformant arrays
390	Undefined pointer
391	Disassociated pointer
392	Corrupt pointer
393	Inconsistent call to FTN95 routine

394	ORDER array should have the same number of elements as SHAPE (RESHAPE)
395	ORDER array is not a permutation of (1,2,...,size(SHAPE)) (RESHAPE)
396	The result array size is different from the size of VECTOR (PACK)
397	The result array size is too small (PACK)
398	MASK is scalar with the value true, the size of VECTOR is less than the size of ARRAY (PACK)
399	MASK is scalar with the value true, the size of ARRAY is different from the size of RESULT (PACK)
400	SHAPE must be rank 1 (RESHAPE)
401	ORDER must be rank 1 (RESHAPE)
402	PAD must not be scalar (RESHAPE)
403	VECTOR has insufficient elements (UNPACK)
404	Result has insufficient elements
405	Reference to undefined character (/FULL_UNDEF)
406	Invalid call to random number seeder
407	DEALLOCATE failed
408	ALLOCATE was unable to obtain sufficient storage
409	Field width zero not allowed on input
410	A function called from within an I/O statement has itself performed I/O
411	This access of date facilities is not Millenium compatible
412	Preconnection unit number not in range 1-99
413	Can't parse preconnection string
414	Pathname is too long
415	Missing closing parenthesis in run-time format
416	Bad request for stack memory
417	Non-conformant array shapes for matrix multiplication (MATMUL)
418	MATMUL can accept, at most, one vector
419	Reference through unset Fortran POINTER

420	Reference through NULL Fortran POINTER
421	Reference through dangling Fortran POINTER
422	Attempt to DEALLOCATE invalid Fortran POINTER
423	Attempt to DEALLOCATE part of a storage block
424	Attempt to DEALLOCATE dangling Fortran POINTER
425	"Integer out of range for conversion to/from unsigned"

22.

Exception handling

Trapping Win32 exceptions

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.

TRAP_EXCEPTION@ and SET_TRAP@ provide the programmer with a method to trap these exception events and to act appropriately. This means that it is possible to trap (say) an underflow event and reset a variable to a known value (say zero).

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 *exceptn.ins* that is provided as part of the compiler 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 the compiler. 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.

The two functions TRAP_EXCEPTION@ and SET_TRAP@ are used to install an event handler for a given event. SET_TRAP@ is used to trap the use of CTRL+C to break into program execution.

Here is a summary of other FTN95 error and exception handling routines that are peculiar to Win32. Details are given in chapter 25.

ACCESS_DETAILS@	Gets details of the access violation.
CLEAR_FLT_UNDERFLOW@	Clears a floating point underflow exception.
EXCEPTION_ADDRESS@	Finds the address of the instruction that generated the exception.
GET_VIRTUAL_COMMON_INFO@	Gets virtual common block details.
PRERR@	Prints the error message associated with a given error code.
RESTORE_DEFAULT_HANDLER@	Removes a user defined exception handler.

TRAP_EXCEPTION@

Purpose To install a user defined exception handler.

Syntax

```
INTEGER*4 FUNCTION TRAP_EXCEPTION@(EXCEPTION, ROUTINE)
INTEGER*4 EXCEPTION, ROUTINE
EXTERNAL ROUTINE
```

Description The function ROUTINE is installed as the handler for the event specified by EXCEPTION.

Return value If EXCEPTION is a valid exception event, the location of the previous handler is returned. 0 is returned if EXCEPTION is an invalid exception code.

The values of EXCEPTION are as follows:

Event	Parameter name	Value
Access violation	ACCESS_VIOLATION	0
Invalid floating point operation	FLT_INVALID_OPERATION	1
Denormal floating point operand	FLT_DENORMAL	2
Floating point divide by zero	FLT_DIV_ZERO	3
Floating point overflow	FLT_OVERFLOW	4
Floating point underflow	FLT_UNDERFLOW	5
Inexact floating point result	FLT_INEXACT_RESULT	6
Floating point stack overflow	FLT_STACK_FAULT	7
Breakpoint	BREAK_POINT	9
Single step	SINGLE_STEP	10
Execution of a privileged instruction	PRIV_INSTRUCTION	11
All exceptions	ALL_EXCEPTIONS	12
All floating point exceptions	ALL_FLOATING_POINT	13
Integer divide by zero	INT_DIVIDEBY_ZERO	14
Down to page reserve	DOWNTO_PAGE_RESERVE	16

Values returned by ROUTINE:

Instruction to event handler	Parameter name	Value
Exception has been handled so continue execution.	CONTINUE_EXECUTION	0
The problem has not been fixed so the system should try to handle this event.	EXCEPTION_UNHANDLED	1
Unable to continue after handling this exception so exit from the program.	NONCONTINUABLE_EXCEPTION	-2

Notes Unless assembler coding is used, ROUTINE should normally return the value NONCONTINUABLE_EXCEPTION. One exception to this rule is floating point underflow for which one can return EXCEPTION_UNHANDLED and then continue.

Integer overflow does not generate an exception unless in /DEBUG mode. In /DEBUG mode the exception can not be trapped.

The above parameter names and values are listed in the file *exceptn.ins*.

Example

```
PROGRAM TrapException
  INCLUDE <exceptn.ins>
  INTEGER TRAP_EXCEPTION@, DivZeroHandler, OldHandler
  EXTERNAL TRAP_EXCEPTION@, DivZeroHandler
  REAL r, zero
  OldHandler=TRAP_EXCEPTION@(FLT_DIV_ZERO, DivZeroHandler)
  zero=0.0
  r=1.234/zero
  PRINT *, 'Normal termination'
END
INTEGER FUNCTION DivZeroHandler()
  INCLUDE <exceptn.ins>
  PRINT *, 'Float divide by zero trapped'
  DivZeroHandler= NONCONTINUABLE_EXCEPTION
END
```

This example illustrates a case where it is not possible to continue executing the program after the event has been trapped. The best that one can do is to save program data and exit cleanly.

SET_TRAP@

Purpose Install a user defined exception handler for Ctrl+C.

Syntax

```
INTEGER*4 SET_TRAP@(ROUTINE, EXCEPTION)
INTEGER*4 EXCEPTION, ROUTINE
EXTERNAL ROUTINE
```

Description EXCEPTION is set to zero to handle CTRL+C. The function ROUTINE is installed as the handler for this event. The location of the previous handler is returned.

If the application is to terminate then ROUTINE should return a zero value. Otherwise a value of 1 is returned.

Example

```
PROGRAM TrapException
  INTEGER SET_TRAP@, OldHandler
  EXTERNAL SET_TRAP@, CtrlcHandler
```

```
  OldHandler=SET_TRAP@(CtrlcHandler, 0)
  DO WHILE(.TRUE.)
    PRINT *, 'Continuing. . .'
  ENDWHILE
  PRINT *, 'Normal termination'
END
INTEGER FUNCTION CtrlcHandler()
  STOP 'Ctrl+C trapped'
  CtrlcHandler=0
END
```

Underflows

Here is a summary of how FTN95 handles underflow. Details appear below. By default FTN95 sets underflow as *unmasked* so that when a underflow occurs an exception is generated. FTN95 handles this exception internally and by default allows the computation to continue. A call to the routine `MASK_UNDERFLOW@` sets underflow as *masked* so that an underflow does not generate an exception. A call of `PERMIT_UNDERFLOW@(.FALSE.)` resets underflow as unmasked and also triggers a run time error when FTN95 handles the exception internally.

Intel architectures use IEEE format floating point numbers. An underflow occurs when the result of a floating point operation is too small to be represented by the floating point format. The discussion below is illustrated using extended precision as an example. Single and double precision underflows are similar.

If x=1E-3000 then y=x*x cannot be represented because the result is 1E-6000 which is smaller than the smallest normalised floating point number (approximately 1E-4932). y would be set to zero.

The floating point unit has a *masked* and an *unmasked* response to underflow. In the above example y is set to zero in both cases. However, the mechanisms are quite different. With the masked response, the actual result is zero and is directly assigned. The unmasked response is to generate an underflow exception which is trapped by the run time library. The failing instruction is decoded and the target register or storage is replaced with zero. Unless `PERMIT_UNDERFLOW@(.FALSE.)` is used, control is then returned to the program at the following instruction and execution continues as normal.

Denormals

Denormals arise for one of two reasons: a) when an integer or an unset variable is passed as an argument that should be an floating point number and b) when the result of a floating point calculation is too small to be a normalised number but still large enough to have a representation (this only occurs when using the masked response to underflow).

In case (b), when the smallest exponent of the number is reached, leading zeroes are introduced into the significand of the number. Thus precision is lost for these numbers. Smaller numbers introduce more leading zeroes and this continues until no more leading zeros can be introduced and the result becomes zero. This process is called *gradual underflow.* Only when the number is less than about 1E-4951 does it become zero.

For example, the hex pattern below is the smallest positive extended precision normalised number. This is defined as such in the IEEE specification.

Exponent	Significand
0001	8000000000000000 (64 bits of precision)

Note that the most significant bit of the significand is 1 (8 is 1000 in binary). This is the *normalisation* bit. Every valid floating point number is shifted so that this bit is 1. This allows 64 bits of precision because there are no leading zeroes. However, if a smaller number is generated, the exponent is 0 and a leading zero is introduced into the significand:

Exponent	Significand
0000	4000000000000000 (63 bits of precision)

The number is then termed a *denormal*. The smallest denormal is:

Exponent	Significand
0000	0000000000000001 (1 bit of precision)

Although precision is lost for these extremely small values, the unmasked response of replacing them with zero represents a total loss of precision. Denormals, therefore give better numerical results although at times they may be non-zero when zero was expected. Denormals are handled like any other number and are printed correctly by the I/O system.

If underflows become unmasked, an attempt to use a denormal results in a run time error whilst the output field for a denormal is filled with "?".

Masking underflows

Underflows can be masked by calling the subroutine `MASK_UNDERFLOW@`

```
SUBROUTINE MASK_UNDERFLOW@()
```

Underflows can be unmasked by calling the subroutine UNMASK_UNDERFLOW@

```
SUBROUTINE UNMASK_UNDERFLOW@()
```

This is the default. If one of these is used, it should be the first executable line in your program. You are very strongly advised not to change the underflow state after this. Calling PERMIT_UNDERFLOW@(.FALSE.) causes underflows to become unmasked.

The environment variable SALFENVAR can be set to MASK_UNDERFLOW or UNMASK_UNDERFLOW in order to provide a default state for your programs. If there is no environment variable setting, then the default is to unmask underflows.

Summary

Masking underflows provides a very safe way of handling underflows and achieving better numerical results, although you should be aware of the issues involved.

The table below shows the range of positive floating point values.

	Largest	**Smallest**	**Smallest denormal** (masked underflows)
Single precision	1E38	1E-38	1E-46
Double precision	1E308	1E-308	1E-324
Extended precision	1E4932	1E-4932	1E-4951

23.

Intrinsic functions

Overview

This chapter provides details of the intrinsic functions that are part of the Fortran 95 standard. The Salford intrinsic called LOC is described together with the Salford CORE intrinsics on page 135.

Array processes

ALL	Tests if all the elements in a logical array are true.
ANY	Tests if any of the elements in a logical array are true.
COUNT	Counts the number of true elements of MASK along dimension DIM.
CSHIFT	Performs a circular shift on an array expression.
DOT_PRODUCT	Gets the dot-product of two numeric or logical vectors.
EOSHIFT	Performs an end-off shift on an array expression.
LBOUND	Returns all the lower bounds or a specified lower bound of an array.
MATMUL	Multiplies two numeric or logical matrices.
MAXLOC	Gets the first position in an array where the element has the maximum value for the array.
MAXVAL	Gets the maximum value of the elements of an array.
MERGE	Merges two arrays according to a given mask.
MINLOC	Gets the first position in an array where the element has the minimum value for the array.
MINVAL	Gets the minimum value of the elements of an array.
PACK	Packs a given array into an array of rank one.
PRODUCT	Gets the product of all the elements of an array.
RESHAPE	Constructs an array of a specified shape from the elements of a given array.

SHAPE	Gets the shape of an array or a scalar.
SIZE	Gets the extent of an array along a specified dimension or the total number of elements in the array.
SPREAD	Replicates an array by adding a dimension.
SUM	Sums the elements of an array.
TRANSPOSE	Transposes an array of rank two.
UBOUND	Gets the upper bound or upper bounds for the indexes of an array.
UNPACK	Unpacks a one-dimensional array under the control of a mask.

Bitwise operations

BIT_SIZE	Gets the number of bits in an integer.
BTEST	Tests if a particular bit in an integer is set.
IAND	Performs a bitwise AND operation.
IBCLR	Clears a single bit.
IBITS	Extracts a sequence of bits.
IBSET	Sets a single bit.
IEOR	Performs a bitwise exclusive OR operation.
IOR	Performs a bitwise OR operation.
ISHFT	Performs a bitwise shift operation.
ISHFTC	Performs a bitwise circular shift operation.
MVBITS	Copies a sequence of bits.
NOT	Performs a bitwise NOT operation.

Mathematical functions

ACOS	Gets the arccosine (inverse cosine) of the argument.
ASIN	Gets the arcsine (inverse sine) of the argument.
ATAN	Gets the arctangent (inverse tangent) of the argument.
ATAN2	Gets the arctangent (inverse tangent) from the arguments.
COS	Gets the cosine of the argument.
COSH	Gets the hyperbolic cosine of the argument.
DIM	Gets the difference if positive, otherwise zero.
DPROD	Gets the double precision value of multiplying two reals.
EXP	Gets the exponential of the argument.
LOG	Gets the natural logarithm of the argument.
LOG10	Gets the common logarithm of the argument.
SIGN	Forms a number from the absolute value of one number and the sign of another.

SIN	Gets the sine of the argument.
SINH	Gets the hyperbolic sine of the argument.
SQRT	Gets the square root of the argument.
TAN	Gets the tangent of the argument.
TANH	Gets the hyperbolic tangent of the argument.

Model inquiry functions

EPSILON	Gets the smallest positive value for the given kind type of a real variable.
HUGE	Gets the largest positive value for the current model and the given kind type of a real or an integer variable.
MAXEXPONENT	Gets the maximum exponent for the kind type of the argument.
MINEXPONENT	Gets the minimum exponent for the kind type of the argument.
NEAREST	Gets the nearest available value to the right or to the left.
PRECISION	Gets the decimal precision for real numbers with kind type of the argument.
RADIX	Gets the base used internally for a given type and kind type.
RANGE	Gets the range of the decimal exponent for integer or real numbers with a given kind type.
RRSPACING	Gets the reciprocal of the relative spacing of model numbers near the argument value.
SCALE	Provides a model-based scaling of a real value.
SET_EXPONENT	Sets the exponent of a real value relative to that used by model numbers.
SPACING	Gets the absolute spacing of model numbers near the argument value.
TINY	Gets the smallest positive number of the model representing numbers of the same type and kind type parameter as the argument.

Number and type operations

ABS	Gets the absolute value of the argument.
AIMAG	Gets the imaginary part of a complex number.
AINT	Truncates a REAL number to its integer part.
ANINT	Rounds off a REAL number to its nearest integer value.
CEILING	Gets the least integer greater than or equal to its argument.
CMPLX	Converts a number or a pair of numbers to complex type.
CONJG	Gets the conjugate of a complex number.
DBLE	Converts to double precision.

DIGITS	Gets the number of significant digits for a given type of number.
EXPONENT	Gets the exponent part of a real number.
FLOOR	Gets the greatest integer less than or equal to its argument.
FRACTION	Gets the fractional part (significand) of a real number.
INT	Converts to integer type.
KIND	Gets the value of the kind type of the argument.
LOGICAL	Converts to another logical kind type.
MAX	Gets the maximum value of a list of numbers.
MIN	Gets the minimum value of a list of numbers.
MOD	Gets the remainder after division.
MODULO	Gets the modulo.
NINT	Gets the nearest integer to a given real value.
RANDOM_NUMBER	Gets a pseudorandom number or an array of pseudorandom numbers in a uniform distribution over the range $0 \leq x < 1$
RANDOM_SEED	Restarts or interrogate the pseudorandom number generator used by RANDOM_NUMBER.
REAL	Converts to real type.
SELECTED_INT_KIND	Gets the kind type parameter for integers with a given range.
SELECTED_REAL_KIND	Gets the kind type parameter for real values with a given precision and range.
TRANSFER	Changes the type and/or kind type of a scalar or array.

Storage allocation

ALLOCATED	Determines whether or not an allocatable array is currently allocated.
ASSOCIATED	Determines whether or not a pointer is associated or is associated with a given target.
NULL	Disassociates a pointer.

String operations

ACHAR	Gets a character in the ASCII collating sequence.
ADJUSTL	Removes leading spaces from a character string and insert them at the end.
ADJUSTR	Removes trailing spaces from a character string and insert them at the beginning.
CHAR	Gets the character in a given position of the collating sequence associated with the specified kind type parameter.
IACHAR	Gets the position of a character in the ASCII collating sequence.
ICHAR	Gets the position of a character in the collating sequence of a given

	kind type.
INDEX	Gets the starting position of a substring within a string.
LEN	Gets the length of a character string.
LEN_TRIM	Gets the length of a character string excluding trailing blanks.
LGE	Tests if a string is lexically greater than or equal to another string, using the ASCII collating sequence.
LGT	Tests if a string is lexically greater than another string, using the ASCII collating sequence.
LLE	Tests if a string is lexically less than or equal to another string, using the ASCII collating sequence.
LLT	Tests if a string is lexically less than another string, using the ASCII collating sequence.
REPEAT	Concatenates several copies of a string.
SCAN	Scans a string for any one of the characters in a set of characters.
TRIM	Removes trailing spaces from a string.
VERIFY	Gets the position of the first character in a string that does not appear in a given set of characters.

Subroutine arguments

PRESENT	Determines whether an optional argument is present.

Time and date functions

CPU_TIME	Gets the processor time.
DATE_AND_TIME	Gets the real-time date and clock time.
SYSTEM_CLOCK	Gets integer data from a real-time clock.

ABS

Purpose To get the absolute value of the argument.

Class Elemental function.

Syntax

```
INTEGER FUNCTION ABS(A);        INTEGER A
REAL FUNCTION ABS(A);           REAL A
REAL FUNCTION ABS(A);           COMPLEX A
```

Return value If A is INTEGER or REAL then the return value is A (if A $\geq$ 0) and -A (if A < 0).

If A is COMPLEX and A = (*x*, *y*) then the return value is $\sqrt{x^2 + y^2}$.

ACHAR

Purpose To get a character in the ASCII collating sequence.

Class Elemental function.

Syntax
```
CHARACTER FUNCTION ACHAR(I);  INTEGER I
```

Return value If I is in the range 0 to 255, ACHAR returns the single character of type KIND('A') that is given by the ASCII collating sequence.

Notes IACHAR is the inverse function.

ACOS

Purpose To get the Arccosine (inverse cosine) of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION ACOS(X);          REAL X
```

Description X must be in the range $-1 \leq X \leq 1$.

Return value The result is given in radians in the range $0 \leq result \leq \pi$.

ADJUSTL

Purpose To remove leading spaces from a character string and insert them at the end.

Class Elemental function.

Syntax
```
CHARACTER (LEN=*) FUNCTION ADJUSTL(STRING)
CHARACTER (LEN=*) STRING
```

Return value The result has then same type and length as STRING.

ADJUSTR

Purpose To remove trailing spaces from a character string and insert them at the beginning.

Class Elemental function.

Syntax
```
CHARACTER (LEN=*) FUNCTION ADJUSTR(STRING)
CHARACTER (LEN=*) STRING
```

Return value The result has then same type and length as STRING.

AIMAG

Purpose To get the imaginary part of a complex number.

Class Elemental function.

Syntax
```
REAL FUNCTION AIMAG(Z);        COMPLEX Z
```

Return value If Z = (*x*, *y*) then the result is *y*.

AINT

Purpose To truncate a REAL number to its integer part.

Class Elemental function.

Syntax
```
REAL FUNCTION AINT(A [,KIND]); REAL A; INTEGER KIND
```

Description KIND is optional. If it is omitted the return type is the same as A. If KIND is present it specifies the kind type of the return value.

Return value The argument is truncated towards zero.

Example AINT(1.6) gives 1.0. AINT(-2.6) gives -2.0.

ALL

Purpose To test if all the elements in a logical array are true.

Class Transformational function.

Syntax

```
LOGICAL FUNCTION ALL(MASK [,DIM]);
LOGICAL MASK; INTEGER DIM
```

Description MASK is a logical array. DIM is optional and is a value in the range $1 \leq \text{DIM} \leq n$ where *n* is the rank (i.e. the number of dimensions) of MASK. ALL returns a logical array or scalar whose kind type is the same as an element of MASK. If MASK has rank 1 or if DIM is not present then ALL returns a scalar otherwise ALL returns an array whose rank is one less than that of MASK. If MASK has rank 1 then a supplied value of DIM is ignored.

Return value ALL(MASK) returns the scalar value true if all the elements of MASK are true. It returns false if any element of MASK is false.

If MASK has rank *n* greater than 1 then ALL(MASK, DIM) returns a logical array of rank *n* - 1 whose elements are given by ALL(MASK) applied along dimension DIM.

Example If MASK = $\begin{bmatrix} true, true, false \\ true, true, true \end{bmatrix}$

then ALL(MASK, 1) returns [*true*, *true*, *false*]

whilst ALL(MASK, 2) returns [*false*, *true*]

ALLOCATED

Purpose To determine whether or not an allocatable array is currently allocated.

Class Inquiry function.

Syntax

```
LOGICAL FUNCTION ALLOCATED(ARRAY)
```

Description ARRAY is any allocatable array.

Return value Using the default LOGICAL type, the result is true if the array is currently allocated, otherwise false.

ANINT

Purpose To round off a REAL number to its nearest integer value

Class Elemental function.

Syntax

```
REAL FUNCTION ANINT(A [,KIND])
REAL A; INTEGER KIND
```

Description KIND is optional. If it is omitted the return type is the same as A. If KIND is present it specifies the kind type of the return value.

Return value The argument is rounded to its nearest integer value.

Example ANINT(1.6) gives 2.0. ANINT(-2.6) gives -3.0.

ANY

Purpose To test if any of the elements in a logical array are true.

Class Transformational function.

Syntax
```
LOGICAL FUNCTION ANY(MASK [,DIM])
LOGICAL MASK; INTEGER DIM
```

Description MASK is a logical array. DIM is optional and is a value in the range $1 \leq DIM \leq n$ where n is the rank (i.e. the number of dimensions) of MASK. ANY returns a logical array or scalar whose kind type is the same as an element of MASK. If MASK has rank 1 or if DIM is not present then ANY returns a scalar otherwise ANY returns an array whose rank is one less than that of MASK. If MASK has rank 1 then a supplied value of DIM is ignored.

Return value ANY(MASK) returns the scalar value true if any of the elements of MASK are true. It returns false if all of the elements of MASK is false.

If MASK has rank n greater than 1 then ANY(MASK, DIM) returns a logical array of rank n - 1 whose elements are given by ANY(MASK) applied along dimension DIM.

Example If MASK = $\begin{bmatrix} true, & false, & false \\ true, & true, & false \end{bmatrix}$

then ANY(MASK, 1) returns [*true*, *true*, *false*]

whilst ANY(MASK, 2) returns [*true*, *true*]

ASIN

Purpose To get the Arcsine (inverse sine) of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION ASIN(X);          REAL X
```

Description X must be in the range $-1 \leq X \leq 1$.

Return value The result is given in radians in the range $-\pi/2 \leq result \leq \pi/2$.

ASSOCIATED

Purpose To determine whether or not a pointer is associated or is associated with a given target.

Class Inquiry function.

Syntax
```
LOGICAL FUNCTION ASSOCIATED(POINTER [,TARGET])
```

Description POINTER is a pointer of any type whose status has been set as *associated* or *disassociated* (not *undefined*). TARGET is optional and is a pointer or a target. TARGET has the same type, type parameters, and rank as POINTER. If it is a pointer then its status must not be *undefined*.

Return value The ISO standard defines the result as:

1. If TARGET is absent, the result is true if POINTER is currently associated with a target and false if it is not.
2. If TARGET is present and is a scalar target, the result is true if TARGET is not a zero-sized storage sequence and the target associated with POINTER occupies the same storage units as TARGET. Otherwise, the result is false. If POINTER is disassociated, the result is false.
3. If TARGET is present and is an array target, the result is true if the target associated with POINTER and TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If POINTER is disassociated, the result is false.
4. If TARGET is present and is a scalar pointer, the result is true if the target associated with POINTER and the target associated with TARGET are not zero-sized storage sequences and they occupy the same storage units. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.
5. If TARGET is present and is an array pointer, the result is true if the target associated with POINTER and the target associated with TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.

ATAN

Purpose To get the Arctangent (inverse tangent) of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION ATAN(X);          REAL X
```

Description X can be any real value.

Return value The result is given in radians in the range $-\pi/2 \leq result \leq \pi/2$.

ATAN2

Purpose To get the Arctangent (inverse tangent) from the arguments.

Class Elemental function.

Syntax
```
REAL FUNCTION ATAN2(Y,X);       REAL X, Y
```

Description (X, Y) are cartesian coordinates and the result is the value of Arctangent(Y/X). If Y>0, the result is positive. If Y=0, the result is zero if X>0 and π if X<0. If Y<0, the result is negative. If X=0, the absolute value of the result is $\pi/2$.

The arguments must not both have the value zero.

Return value The result is given in radians in the range $-\pi < result \leq \pi$.

BIT_SIZE

Purpose To get the number of bits in an integer.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION BIT_SIZE(I); INTEGER I
```

Description I can be a scalar or an array.

Return value The result is a scalar of the same type as the argument.

BTEST

Purpose To test if a particular bit in an integer is set.

Class Elemental function.

Syntax
```
LOGICAL FUNCTION BTEST(I, POS);     INTEGER I, POS
```

Description POS must be in the range $0 \leq POS < BIT\_SIZE(I)$. Bit zero is the least significant bit. At least one of I and POS must be a scalar.

Return value The result has the default logical type.

CEILING

Purpose To get the least integer greater than or equal to its argument.

Class Elemental function.

Syntax
```
INTEGER FUNCTION CEILING (A [,KIND])
REAL A; INTEGER KIND
```

Description KIND is optional. If it is omitted the return type is the default integer type. If KIND is present it specifies the kind type of the return value.

Example CEILING(-2.6) has the integer value -2.

CHAR

Purpose To get the character in a given position of the collating sequence associated with the specified kind type parameter.

Class Elemental function.

Syntax
```
CHARACTER FUNCTION CHAR(I [,KIND]); INTEGER I, KIND
```

Description I represents the position in the collating sequence numbering from zero. KIND is optional and can be used to specify the (scalar) kind type parameter of the character set. If KIND is not present the default character set is used. CHAR is the inverse of the intrinsic function ICHAR.

Return value The result is a character of length one.

Example CHAR(97) gives ‘a’ when used with the ASCII collating sequence.

CMPLX

Purpose To convert a number or a pair of numbers to complex type.

Class Elemental function.

Syntax
```
COMPLEX FUNCTION CMPLX(X [,Y,KIND]);REAL X;   INTEGER KIND
COMPLEX FUNCTION CMPLX(X [,Y,KIND]);INTEGER X;INTEGER KIND
COMPLEX FUNCTION CMPLX(X [,KIND]);  COMPLEX X;INTEGER KIND
```

Description KIND an optional scalar value. If it is omitted the return type is the default complex type. If KIND is present it specifies the kind type of the return value.

If X is real or integer it represents the real part of the complex result. In this case if Y is absent its default value is zero. Y is optional. It is either real or integer (not complex) and represents the imaginary part of the complex result.

Example CMPLX(-4) gives the value (-4.0, 0.0).

CONJG

Purpose To get the conjugate of a complex number.

Class Elemental function.

Syntax
```
COMPLEX FUNCTION CONJG(Z);          COMPLEX Z
```

Return value The result has the same type as Z.

Example CONJG ((1.0, 2.0)) has the value (1.0, -2.0).

COS

Purpose To get the Cosine of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION COS(X);          REAL X
COMPLEX FUNCTION COS(X);       COMPLEX X
```

Description If X is real it is an angle measured in radians. If X is complex its real part is in radians.

COSH

Purpose To get the Hyperbolic cosine of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION COSH(X);          REAL X
```

Description The result has the same type as X.

COUNT

Purpose Count the number of true elements of MASK along dimension DIM.

Class Transformational function.

Syntax
```
INTEGER FUNCTION COUNT(MASK [,DIM])
LOGICAL MASK; INTEGER DIM
```

Description MASK is a logical array. DIM is optional and is a value in the range $1 \leq \text{DIM} \leq n$ where n is the rank (i.e. the number of dimensions) of MASK. COUNT returns an integer array of the default kind type. If MASK has rank 1 or if DIM is not present then COUNT returns a scalar otherwise COUNT returns an array whose rank is one less than that of MASK. If MASK has rank 1 then a supplied value of DIM is ignored.

Return value ANY(MASK) returns the scalar value true if any of the elements of MASK are true. It returns false if all of the elements of MASK is false.

If MASK has rank n greater than 1 then ANY(MASK, DIM) returns a logical array of rank n - 1 whose elements are given by ANY(MASK) applied along dimension DIM.

Example If MASK = $\begin{bmatrix} true, & false, & false \\ true, & true, & false \end{bmatrix}$

then COUNT(MASK, 1) returns [2, 1, 0]

whilst COUNT(MASK, 2) returns [1, 2]

CPU_TIME

Purpose To get the processor time.

Class Subroutine.

Syntax
```
SUBROUTINE CPU_TIME(TIME);     REAL,INTENT(OUT)::TIME
```

Return value TIME is a scalar value giving the processor time in seconds. A negative value denotes an error condition.

Example
```
REAL TIME1, TIME2
CALL CPU_TIME(TIME1)
! Processing...
CALL CPU_TIME(TIME2)
PRINT *, 'Processing time was ', TIME2-TIME1, ' seconds'
```

CSHIFT

Purpose Perform a circular shift on an array expression.

Class Transformational function.

Syntax
```
FUNCTION CSHIFT(ARRAY, SHIFT [, DIM])
INTEGER SHIFT
INTEGER DIM
```

Description Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.

DIM is optional and is a value in the range $1 \leq \text{DIM} \leq n$ where n is the rank (i.e. the number of dimensions) of ARRAY. The default value of DIM is 1.

Return value The result has the same type, type parameters and shape as ARRAY.

Example If A = [1, 2, 3, 4, 5] then CSHIFT(A, 2) (i.e. shifted to the left) gives [3, 4, 5, 1, 2], and CSHIFT(A, -2) gives [4, 5, 1, 2, 3].

If B = $\begin{bmatrix}1,2,3\\4,5,6\\7,8,9\end{bmatrix}$ then CSHIFT(B, 1, 2) gives $\begin{bmatrix}2,3,1\\5,6,4\\8,9,7\end{bmatrix}$.

If SHIFT = [1, 0, 2] then CSHIFT(B, SHIFT, 2) gives $\begin{bmatrix}2,3,1\\4,5,6\\9,7,8\end{bmatrix}$.

DATE_AND_TIME

Purpose To get the real-time date and clock time.

Class Subroutine

Syntax

```
SUBROUTINE DATE_AND_TIME([DATE, TIME, ZONE, VALUES])
CHARACTER (LEN=*),INTENT(OUT)::DATE,TIME,ZONE
INTEGER,INTENT(OUT)::VALUES(8)
```

Description All the arguments are optional.

The leftmost 8 characters of DATE are assigned a value of the form CCYYMMDD, where CC is the century, YY the year , MM the month, and DD the day.

The leftmost 10 characters of TIME are assigned a value of the form hhmmss.sss, where hh is the hour, mm is the minutes, and ss.sss is the seconds and milliseconds.

The leftmost 5 characters of ZONE are assigned a value of the form ±hhmm, where hh and mm are the time difference in hours and minutes with respect to Coordinated Universal Time (UTC i.e. GMT).

VALUES(1) is the year (e.g. 1997).

VALUES(2) is the number of the month (1 denotes January).

VALUES(3) is the day of the month in the range 1 to 31.

VALUES(4) is the time difference in minutes with respect to UTC.

VALUES(5) is the hour in the range 0 to 23.

VALUES(6) is the minutes in the range 0 to 59.

VALUES(7) is the seconds in the range 0 to 59.

VALUES(8) is the milliseconds in the range 0 to 999.

Return value On error character variables are returns as blanks and integer values are returned as -

HUGE(0).

DBLE

Purpose To convert to double precision.

Class Elemental function.

Syntax
```
DOUBLE PRECISION FUNCTION DBLE(A)
```

Description A is of type INTEGER, REAL or COMPLEX. If A is COMPLEX the result is the real part of A. The kind type of the result is KIND(0.0D0).

DIGITS

Purpose To get the number of significant digits for a given type of number.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION DIGITS(X)
```

Description X is integer or real, scalar or array. The result is dependent on the type and kind type of the variable. It is the number of significant digits in the model for an integer and the number of significant figures in the mantissa of the model for a real. The result depends on the base that is used in the model (e.g. base 2 or base 10).

DIM

Purpose To get the difference if positive, otherwise zero.

Class Elemental function.

Syntax
```
INTEGER FUNCTION DIM(X,Y);    INTEGER X,Y
REAL    FUNCTION DIM(X,Y);    REAL X,Y
```

Description Both the result an Y have the same type and kind type as X.

Return value X-Y if X>Y, otherwise zero.

DOT_PRODUCT

Purpose To get the dot-product of two numeric or logical vectors.

Class Transformational function.

Syntax
```
FUNCTION DOT_PRODUCT(VECTOR_A, VECTOR_B)
```

Description The type of the result is a scalar numeric (integer, real, or complex) or logical type and is the same type as each element of the one-dimensional array VECTOR_A.

VECTOR_A and VECTOR_B are one-dimensional arrays with the same length. VECTOR_A is of numeric type or logical type. If VECTOR_A is numeric then VECTOR_B must be numeric (though perhaps not the same type) and if VECTOR_A is logical then VECTOR_B must be logical.

Return value If VECTOR_A is integer or real then the result is the inner product of the two arrays (i.e. the sum of the products of corresponding elements).

If VECTOR_A is complex then its conjugate is used in the inner product and the result is the complex value that is the sum of the terms. If VECTOR_B = VECTOR_A then the imaginary part of the result will be approximately zero.

If VECTOR_A is logical then the result is the same as ANY(VECTOR_A .AND. VECTOR_B).

If the vectors have size zero then the result is zero or false.

DPROD

Purpose To get the double precision value of multiplying two reals.

Class Elemental function.

Syntax
```
DOUBLE PRECISION FUNCTION DPROD(X, Y);  REAL X,Y
```

EOSHIFT

Purpose Perform an end-off shift on an array expression.

Class Transformational function.

Syntax
```
FUNCTION EOSHIFT(ARRAY, SHIFT [, BOUNDARY, DIM])
INTEGER SHIFT
```

```
INTEGER DIM
```

Description Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.

BOUNDARY is optional and takes the following default values depending in its type:

Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character	filled with blanks

DIM is optional and is a value in the range $1 \leq \text{DIM} \leq n$ where n is the rank (i.e. the number of dimensions) of ARRAY. The default value of DIM is 1.

Return value The result has the same type, type parameters and shape as ARRAY.

Example If A = [1, 2, 3, 4, 5] then EOSHIFT(A, 2) (i.e. shifted to the left) gives [3, 4, 5, 0, 0], and EOSHIFT(A, -2, 6) gives [6, 6, 1, 2, 3].

If $A = \begin{bmatrix} 1,2,3 \\ 4,5,6 \\ 7,8,9 \end{bmatrix}$ then EOSHIFT(A, 1, DIM=2) gives $\begin{bmatrix} 2,3,0 \\ 5,6,0 \\ 8,9,0 \end{bmatrix}$.

If S = [1, 0, 2] and B = [10, 0, 0] then EOSHIFT(A, S, BOUNDARY=B, DIM=2) gives $\begin{bmatrix} 2,3,10 \\ 4,5,6 \\ 9,0,0 \end{bmatrix}$.

EPSILON

Purpose To get the smallest positive value for the given kind type of a real variable.

Class Inquiry function.

Syntax

```
REAL FUNCTION EPSILON(X);      REAL X;
```

Description If p is the number of significant digits (i.e. p= DIGITS(X)) and b is the base that is used in the model (e.g. base 2 or base 10) then EPSILON(X) = b^{1-p}.

Return value X may be a scalar or an array but the result is a scalar.

EXP

Purpose To get the exponential of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION EXP(X);          REAL X;
COMPLEX FUNCTION EXP(X);       COMPLEX X;
```

Description The result is e^x and has the same type as X.

EXPONENT

Purpose To get the exponent part of a real number.

Class Elemental function.

Syntax
```
INTEGER FUNCTION EXPONENT(X);        REAL X;
```

Return value The result is zero if X is zero otherwise the result is the exponent part of the model used for real numbers (i.e. the power to which the base for the model is raised).

FLOOR

Purpose To get the greatest integer less than or equal to its argument.

Class Elemental function.

Syntax
```
INTEGER FUNCTION FLOOR(A [, KIND])
REAL A; INTEGER KIND
```

Description KIND is optional. If it is omitted the return type is the default integer type. If KIND is present it specifies the kind type of the return value.

Return value FLOOR(2.6) gives 2. FLOOR(-3.4) gives -4.

FRACTION

Purpose To get the factional part (mantissa) of a real number.

Class Elemental function.

Syntax `REAL FUNCTION FRACTION(X);  REAL X;`

Return value The result is zero if X is zero otherwise the result is the fractional part of the model used for real numbers. It has the same kind type as the argument.

HUGE

Purpose To get the largest positive value for the current model and the given kind type of a real or an integer variable.

Class Elemental function.

Syntax
```
REAL    FUNCTION HUGE(X);  REAL X
INTEGER FUNCTION HUGE(X);  INTEGER X
```

Return value X may be a scalar or an array but the result is a scalar.

IACHAR

Purpose To get the position of a character in the ASCII collating sequence.

Class Elemental function.

Syntax `INTEGER FUNCTION IACHAR(C);  CHARACTER C`

Description C is a character of length one.

Return value The result is an integer in the range $0 \leq result \leq 127$. Its type is the default integer type.

Example IACHAR(‘a’) gives 97.

IAND

Purpose To perform a bitwise AND operation.

Class Elemental function.

Syntax `INTEGER FUNCTION IAND(I, J);  INTEGER I, J`

Description J and the result have the same kind type as I.

Return value Each bit of I is ANDed with the bit of J in the same position according to the table:

I	J	IAND
1	1	1
1	0	0
0	1	0
0	0	0

IBCLR

Purpose To clear a single bit.

Class Elemental function.

Syntax
```
INTEGER FUNCTION IBCLR(I, POS);     INTEGER I, POS
```

Description The position POS is non-negative with zero as the least significant bit. The result has the same kind type as I.

Return value The result is the same as I except that bit POS is zero.

IBITS

Purpose To extract a sequence of bits.

Class Elemental function.

Syntax
```
INTEGER FUCNTION IBITS(I, POS, LEN);  INTEGER I, POS, LEN
```

Description The position POS is non-negative with zero as the least significant bit. The length of the sequence LEN is non-negative. The result has the same kind type as I.

Return value The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted with all other bits zero.

IBSET

Purpose To set a single bit.

Class Elemental function.

Syntax
```
INTEGER FUNCTION IBSET(I, POS);     INTEGER I, POS
```

Description The position POS is non-negative with zero as the least significant bit. The result has the same kind type as I.

Return value The result is the same as I except that bit POS is one.

ICHAR

Purpose To get the position of a character in the collating sequence of a given kind type.

Class Elemental function.

Syntax
```
INTEGER FUNCTION ICHAR(C);    CHARACTER C
```

Description C is a character of length one.

Return value The result is an integer in the range $0 \leq result \leq n - 1$ where n is the number of characters in the collating sequence. Its type is the default integer type.

IEOR

Purpose To perform a bitwise exclusive OR operation.

Class Elemental function.

Syntax
```
INTEGER FUNCTION IEOR(I, J);  INTEGER I, J
```

Description J and the result have the same kind type as I.

Return value Each bit of I is XORed with the bit of J in the same position according to the table:

I	J	IEOR
1	1	0
1	0	1
0	1	1
0	0	0

INDEX

Purpose To get the starting position of a substring within a string.

Class Elemental function.

Syntax
```
INTEGER FUNCTION INDEX(STRING, SUBSTRING [, BACK])
CHARACTER (LEN=*) STRING, SUBSTRING
LOGICAL BACK
```

Description If BACK is omitted or has the value false then STRING is scanned from the beginning for the first occurrence of SUBSTRING. Zero is returned if all of SUBSTRING is not found or if LEN(SUBSTRING) = 0.

If BACK is true then scanning is from the end to find the last occurrence. Zero is returned if all of SUBSTRING is not found. LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Return value If SUBSTRING is found the result is a positive integer giving its position.

Example INDEX('SALFORD', 'FORD') gives 4.
INDEX('FORTRAN', 'R', BACK=.TRUE.) gives 5.

INT

Purpose To convert to integer type.

Class Elemental function.

Syntax
```
INTEGER FUNCTION INT(A [,KIND]);  INTEGER KIND
```

Description A is of type INTEGER, REAL or COMPLEX. The kind type of the result is KIND if it is present otherwise it is the default integer kind.

Return value If A is integer the result is A (if it is in range).

If A is real the result is the integer part of A truncated towards zero.

If A is complex, INT(A) is obtained by applying the above rule to the real part of A.

IOR

Purpose To perform a bitwise OR operation.

Class Elemental function.

Syntax
```
INTEGER FUNCTION IOR(I, J);   INTEGER I, J
```

Description J and the result have the same kind type as I.

Return value Each bit of I is ORed with the bit of J in the same position according to the table:

I	J	IEOR
1	1	1
1	0	1
0	1	1
0	0	0

ISHFT

Purpose To perform a bitwise shift operation.

Class Elemental function.

Syntax
```
INTEGER FUNCTION ISHFT(I, SHIFT);   INTEGER I, SHIFT
```

Description SHIFT is a positive or negative integer whose absolute value is ≤ BIT_SIZE(I). A positive value denotes a shift to the left. Bits that are shifted out are lost and zeros are shifted in from the opposite end. The result has the same kind type as I.

ISHFTC

Purpose To perform a bitwise circular shift operation.

Class Elemental function.

Syntax
```
INTEGER FUNCTION ISHFTC(I, SHIFT [, SIZE])
INTEGER I, SHIFT, SIZE
```

Description SHIFT is a positive or negative integer whose absolute value is ≤ SIZE.

SIZE is an optional positive integer ≤ BIT_SIZE (I). The default value of SIZE is BIT_SIZE(I).

A positive value of SHIFT denotes a shift to the left. The leftmost SIZE bits are processed and SHIFT bits are shifted out from one end and shifted in from the opposite end. The result has the same kind type as I.

KIND

Purpose To get the value of the kind type of the argument.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION KIND(X)
```

Description X is any intrinsic type (scalar or array).

Return value The result is the kind type parameter of X represented as a scalar of the default integer type.

LBOUND

Purpose To return all the lower bounds or a specified lower bound of an array.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION LBOUND(ARRAY [, DIM])
INTEGER DIM
```

Description ARRAY is an array of any type, not a scalar. If it is a pointer it must be associated. If it is an allocatable array it must be allocated. DIM (optional) is a scalar integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

The result is of default integer type. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Return value

1) For an array section or for an array expression other than a whole array or array structure component, LBOUND(ARRAY, DIM) has the value 1. For a whole array or array structure component, LBOUND(ARRAY, DIM) has the value:
   a) equal to the lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have extent zero or if ARRAY is an assumed-size array of rank DIM, or
   b) 1 otherwise.
2) LBOUND(ARRAY) has a value whose ith component is equal to LBOUND(ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Example If A is declared by the statement

REAL A(2:3, 7:10)

then LBOUND(A) is [2, 7] and LBOUND(A, DIM=2) is 7.

LEN

Purpose To get the length of a character string.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION LEN(STRING)
CHARACTER (LEN=*) STRING
```

Return value The result is the number of characters in STRING if it is scalar or in an element of STRING if it is array of character strings.

Example If STRING is declared as

CHARACTER (LEN=20)::STRING(10)

LEN(STRING) has the value 20.

LEN_TRIM

Purpose To get the length of a character string excluding trailing blanks.

Class Elemental function.

Syntax
```
INTEGER FUNCTION LEN_TRIM(STRING)
CHARACTER (LEN=*) STRING
```

Example `LEN_TRIM('A B  ')` has the value 3 and `LEN_TRIM('  ')` has the value 0.

LGE

Purpose To test if a string is lexically greater than or equal to another string, using the ASCII collating sequence.

Class Elemental function.

Syntax
```
LOGICAL FUNCTION LGE(STRING_A, STRING_B)
CHARACTER (LEN=*) STRING_A, STRING_B
```

Description If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

Return value The result is true if the strings are equal or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false.

The result is true if both strings are of zero length.

LGT

Purpose To test if a string is lexically greater than another string, using the ASCII collating sequence.

Class Elemental function.

Syntax

```
LOGICAL FUNCTION LGT(STRING_A, STRING_B)
CHARACTER (LEN=*) STRING_A, STRING_B
```

Description If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

Return value The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false.

The result is false if both strings are of zero length.

LLE

Purpose To test if a string is lexically less than or equal to another string, using the ASCII collating sequence.

Class Elemental function.

Syntax

```
LOGICAL FUNCTION LLE(STRING_A, STRING_B)
CHARACTER (LEN=*) STRING_A, STRING_B
```

Description If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

Return value The result is true if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false.

The result is true if both strings are of zero length.

LLT

Purpose To test if a string is lexically less than another string, using the ASCII collating sequence.

Class Elemental function.

Syntax
```
LOGICAL FUNCTION LLT(STRING_A, STRING_B)
CHARACTER (LEN=*) STRING_A, STRING_B
```

Description If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

Return value The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false.

The result is false if both strings are of zero length.

LOG

Purpose To get the natural logarithm of the argument.

Class Elemental function.

Syntax
```
REAL    FUNCTION LOG(X);  REAL X
COMPLEX FUNCTION LOG(X);  COMPLEX X
```

Description If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Return value The result has the value $\log_e X$. A complex result is the principal value with imaginary part w in the range $-\pi < w \leq \pi$. 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.

LOG10

Purpose To get the common logarithm of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION LOG(X);  REAL X
```

Description X must be greater than zero.

Return value The result has the value $\log_{10} X$.

LOGICAL

Purpose To convert to another logical kind type.

Class Elemental function.

Syntax
```
LOGICAL FUNCTION LOGICAL(L [, KIND])
LOGICAL L
INTEGER KIND
```

Description The kind type of the result is KIND if it is present otherwise it is the default logical kind.

MATMUL

Purpose To multiply two numeric or logical matrices

Class Transformational function.

Syntax
```
FUNCTION MATMUL(MATRIX_A, MATRIX_B)
```

Description MATRIX_A shall be of numeric type (integer, real, or complex) or of logical type. MATRIX_A and MATRIX_B must be array valued and of rank one or two.

MATRIX_B must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Return value If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments. If the arguments are of type logical, the result is of type logical. The result is the standard matrix product of the two matrices. In the case where the numeric result for element (*i* , *j*) takes the form SUM (MATRIX_A (*i*, :) * MATRIX_B (:, *j*)), the corresponding logical result takes the form ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:, *j*)).

The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape (*n*, *m*) and MATRIX_B has shape (*m*, *k*), the result

has shape (*n*, *k*).

Case (ii): If MATRIX_A has shape (*m*) and MATRIX_B has shape (*m*, *k*), the result has shape (*k*).

Case (iii): If MATRIX_A has shape (*n*, *m*) and MATRIX_B has shape (*m*), the result has shape (*n*).

MAX

Purpose To get the maximum value of a list of numbers.

Class Elemental function.

Syntax
```
INTEGER FUNCTION MAX(A1, A2 [, A3, ...]); INTEGER A1,...
REAL    FUNCTION MAX(A1, A2 [, A3, ...]); REAL    A1,...
```

Description The arguments must all be of same type and kind type which will be the type and kind type of the result.

Return value The result is the largest item in the list.

MAXEXPONENT

Purpose To get the maximum exponent for the kind type of the argument.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION MAXEXPONENT(X);  REAL X
```

Description X can be a real scalar or real array of any kind type.

Return value The result is the largest exponent that is permissible for the given kind type.

MAXLOC

Purpose To get the first position in an array where the element has the maximum value for the array.

Class Transformational function.

Syntax
```
INTEGER FUNCTION MAXLOC(ARRAY, DIM [, MASK])
INTEGER FUNCTION MAXLOC(ARRAY [, MASK])
```

```
INTEGER DIM
LOGICAL MASK
```

Description ARRAY is an integer or real array. DIM is a scalar integer in the range $1 \leq DIM \leq n$ where *n* is the rank of ARRAY. MASK is a logical array that is conformable with ARRAY.

Return value This function determines the location of the first element of ARRAY along dimension DIM having the maximum value of the elements identified by MASK.

If DIM is not present, the result is an array of rank one and size *n*, otherwise the result is of rank *n* - 1 and has a shape that is the same as that of ARRAY but omitting dimension DIM.

Example If A = [3, 2, 9, 9] then MAXLOC(A) is 3.

If A = $\begin{bmatrix} 1,2,3,4 \\ 5,6,7,8 \\ 0,0,0,9 \end{bmatrix}$ then MAXLOC(A, MASK = A .LT. 5) is [1, 4] which is the location of the largest value for this mask.

If A = $\begin{bmatrix} 1,2,3,8 \\ 5,6,7,4 \end{bmatrix}$ then MAXLOC(A, DIM = 1) is [2, 2, 2, 1] and MAXLOC(A, DIM = 2) is [4, 3]

MAXVAL

Purpose To get the maximum value of the elements of an array.

Class Transformational function.

Syntax

```
FUNCTION MAXVAL(ARRAY, DIM [, MASK])
FUNCTION MAXVAL(ARRAY [, MASK])
INTEGER DIM
LOGICAL MASK
```

Description ARRAY is an integer or real array. DIM is a scalar integer in the range $1 \leq DIM \leq n$ where *n* is the rank of ARRAY. MASK is a logical array that is conformable with ARRAY.

Return value This function determines the maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK. The result is a scalar or array having the same type.

If DIM is not present, the result is scalar, otherwise the result is of rank *n* - 1 and has a shape that is the same as that of ARRAY but omitting dimension DIM.

If ARRAY has size zero, MAXVAL(ARRAY) gives the negative number of largest magnitude for the given type and kind type.

If MASK is false for all elements then MAXVAL(ARRAY, MASK=MASK) gives the negative number of largest magnitude for the given type and kind type.

Example If A = [3, 2, 9, 9] then MAXVAL(A) is 9.

If A = $\begin{bmatrix} 1,2,3,4 \\ 5,6,7,8 \\ 0,0,0,9 \end{bmatrix}$ then MAXVAL(A, MASK = A .LT. 5) is 4.

If A = $\begin{bmatrix} 1,2,3,8 \\ 5,6,7,4 \end{bmatrix}$ then MAXVAL(A, DIM = 1) is [5, 6, 7, 8] and MAXVAL(A, DIM = 2) is [8, 7]

MERGE

Purpose To merge two arrays according to a given mask.

Class Elemental function.

Syntax

```
FUNCTION MERGE(TSOURCE, FSOURCE, MASK)
LOGICAL MASK
```

Description TSOURCE is an array of any type and kind type. FSOURCE and the result have the same type and kind type as TSOURCE. MASK is a logical array that is conformable to TSOURCE, FSOURCE and the result.

Return value For each element, the result is TSOURCE if MASK is true and FSOURCE otherwise.

Example If A = [1, 4, 5], B = [4, 3, 6] then MERGE(A, B, MASK= A < B) gives [1, 3, 5].

MIN

Purpose To get the minimum value of a list of numbers.

Class Elemental function.

Syntax
```
INTEGER FUNCTION MIN(A1, A2 [, A3, ...]); INTEGER A1,...
REAL    FUNCTION MIN(A1, A2 [, A3, ...]); REAL    A1,...
```

Description The arguments must all be of same type and kind type which will be the type and kind type of the result.

Return value The result is the smallest item in the list.

Example MIN(-1.0, 0.0, 2.0) gives -1.0.

MINEXPONENT

Purpose To get the minimum exponent for the kind type of the argument.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION MAXEXPONENT(X);  REAL X
```

Description X can be a real scalar or real array of any kind type.

Return value The result is the smallest exponent (i.e. the negative value with largest magnitude) that is permissible for the given kind type.

MINLOC

Purpose To get the first position in an array where the element has the minimum value for the array.

Class Transformational function.

Syntax
```
INTEGER FUNCTION MINLOC(ARRAY, DIM [, MASK])
INTEGER FUNCTION MINLOC(ARRAY [, MASK])
INTEGER DIM
LOGICAL MASK
```

Description ARRAY is an integer or real array. DIM is a scalar integer in the range $1 \leq DIM \leq n$ where n is the rank of ARRAY. MASK is a logical array that is conformable with ARRAY.

Return value This function determines the location of the first element of ARRAY along dimension DIM having the minimum value of the elements identified by MASK.

If DIM is not present, the result is an array of rank one and size n, otherwise the

result is of rank *n* - 1 and has a shape that is the same as that of ARRAY but omitting dimension DIM.

Example If A = [3, 2, 2, 9] then MINLOC(A) is 2.

If A = $\begin{bmatrix} 1,2,3,4 \\ 5,6,7,8 \\ 0,0,0,9 \end{bmatrix}$ then MINLOC(A, MASK = A .GT. 3) is [1, 4] which is the location of the smallest value for this mask.

If A = $\begin{bmatrix} 1,2,3,8 \\ 5,6,7,4 \end{bmatrix}$ then MINLOC(A, DIM = 1) is [1, 1, 1, 2] and MINLOC(A, DIM = 2) is [1, 4].

MINVAL

Purpose To get the minimum value of the elements of an array.

Class Transformational function.

Syntax

```
FUNCTION MINVAL(ARRAY, DIM [, MASK])
FUNCTION MINVAL(ARRAY [, MASK])
INTEGER DIM
LOGICAL MASK
```

Description ARRAY is an integer or real array. DIM is a scalar integer in the range $1 \leq DIM \leq n$ where *n* is the rank of ARRAY. MASK is a logical array that is conformable with ARRAY.

Return value This function determines the minimum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK. The result is a scalar or array having the same type.

If DIM is not present, the result is scalar, otherwise the result is of rank *n* - 1 and has a shape that is the same as that of ARRAY but omitting dimension DIM.

If ARRAY has size zero, MINVAL(ARRAY) gives the positive number of largest magnitude for the given type and kind type.

If MASK is false for all elements then MINVAL(ARRAY, MASK=MASK) gives the positive number of largest magnitude for the given type and kind type.

Example If A = [3, 2, 2, 9] then MINVAL(A) is 2.

If A = $\begin{bmatrix} 1,2,3,4 \\ 5,9,7,8 \\ 0,0,0,9 \end{bmatrix}$ then MINVAL(A, MASK = A .GT. 5) is 7.

If A = $\begin{bmatrix} 1,2,3,8 \\ 5,6,7,4 \end{bmatrix}$ then MINVAL(A, DIM = 1) is [1, 2, 3, 4] and MINVAL(A, DIM = 2) is [1, 4]

MOD

Purpose To get the remainder after division.

Class Elemental function.

Syntax
```
INTEGER FUNCTION MOD(A, P);  INTEGER A, P
REAL    FUNCTION MOD(A, P);  REAL    A, P
```

Description P and the result have the same type and kind type as A.

Return value If P ≠ 0, the result is the remainder of A after divsion by P given by A - INT(A/P) * P.

Example MOD(4.0, 3.0) gives approximately 1.0. MOD(8, -6) gives 2.

MODULO

Purpose To get the modulo.

Class Elemental function.

Syntax
```
INTEGER FUNCTION MODULO(A, P);  INTEGER A, P
REAL    FUNCTION MODULO(A, P);  REAL    A, P
```

Description P and the result have the same type and kind type as A. P must be non-zero.

Return value If A is of type integer, MODULO (A, P) has the value R such that A = Q × P + R, where Q is an integer. If P > 0 then $0 \leq R < P$. If P > 0 then $P < R \leq 0$.

If A is of type real the result is A – FLOOR (A / P) * P.

Example MODULO(8, 6) gives 2. MODULO(8, -6) gives -4.

MVBITS

Purpose To copy a sequence of bits.

Class Elemental subroutine.

Syntax

```
SUBROUTINE MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)
INTEGER,INTENT(IN)    ::FROM,FROMPOS,LEN,TOPOS
INTEGER,INTENT(INOUT)::TO
```

Description FROMPOS $\geq$ 0 and FROMPOS + LEN $\leq$ BIT_SIZE(FROM).

LEN $\geq$ 0.

TO has the same kind type as FROM.

TOPOS $\geq$ 0 and TOPOS + LEN $\leq$ BIT_SIZE(TO).

Return value After calling the subroutine the value of TO is modified by copying bits from FROM. LEN bits are copied from FROM starting at FROMPOS. The bits are copied to TO starting at TOPOS. FROM and TO are zero-based counting from the least significant bit.

Example If TO = 4 then after a call to MVBITS(12, 2, 2, TO, 0), TO becomes 7.

NEAREST

Purpose To get the nearest available value to the right or to the left.

Class Elemental function.

Syntax

```
REAL FUNCTION NEAREST(X, S);  REAL X, S
```

Description The sign of S provides the direction. A positive value means to the right (towards positive infinity); a negative value means to the left (towards negative infinity). S must not be zero.

Return value The result is the nearest machine representable value to X (that is not equal to X) in the given direction.

NINT

Purpose To get the nearest integer to a given real value.

Class Elemental function.

Syntax
```
INTEGER FUNCTION NINT(A [, KIND]); REAL A; INTEGER KIND
```

Description If KIND is present it gives the kind type of the integer result, otherwise the result is of default integer type.

Return value The result is obtained by rounding off the given real value to the nearest integer. If A > 0, NINT(A) has the value INT(A + 0.5); if A $\leq$ 0, NINT(A) has the value INT(A – 0.5).

NOT

Purpose To perform a bitwise NOT operation.

Class Elemental function.

Syntax
```
INTEGER FUNCTION NOT(I);       INTEGER I
```

Description The result has the same kind type as I.

Return value Each bit of I is transposed according to the table:

I	NOT(I)
1	0
0	1

NULL

Purpose To disassociate a pointer.

Class Transformational function.

Syntax
```
FUNCTION NULL([MOLD])
```

Description If MOLD is present it is a pointer of any type and gives the type of the result. If MOLD is absent then the type of the result matches the target of the assignment.

Return value This function provides an alternative to using the standard subroutine NULLIFY. This functional form can be included with a declaration.

Example REAL, POINTER :: X => NULL( )

PACK

Purpose To pack a given array into an array of rank one.

Class Transformational function.

Syntax
```
FUNCTION PACK(ARRAY, MASK [, VECTOR])
LOGICAL MASK
```

Description ARRAY is an array of any type. MASK is a logical array that is conformable with ARRAY. VECTOR is optional. VECTOR and the result of the function are arrays of rank one with the same type and type parameters as ARRAY.

If VECTOR is absent the size of the result is equal to the number of elements of MASK that are true. If VECTOR is present the size of the result is equal to the size of VECTOR which must not be less than the number of elements of MASK that are true.

Return value The result is a one-dimsional array obtained by selecting those values of ARRAY for which the corresponding element in MASK is true. VECTOR can be used to provide default values for the result after the point at which packing ceases.

Example If M = $\begin{bmatrix} 1,2,3 \\ 4,5,6 \end{bmatrix}$ then PACK(M, MASK=M>2) gives [4, 5, 3, 6].

If also V=[0, 0, 0, 0, 0] then PACK(M, MASK=M>2, VECTOR=V) gives [4, 5, 3, 6, 0].

PRECISION

Purpose To get the decimal precision for real numbers with kind type of the argument.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION PRECISION(X); REAL X
INTEGER FUNCTION PRECISION(X); COMPLEX X
```

Description X may real real or complex, scalar or array valued. The result is a scalar integer of the default kind type.

Return value The result is the number of significant decimal figures that can be stored for the given real kind type.

PRESENT

Purpose To determine whether an optional argument is present.

Class Inquiry function.

Syntax
```
LOGICAL FUNCTION PRESENT(A)
```

Description The argument A is the name of an optional dummy argument in the current subprogram. It may be of any type, a scalar or an array, and it may be a pointer. It may also be the (dummy) name of a procedure. There is no assumed INTENT attribute for A.

Return value The result has the value true if the optional argument A is present in the current call to the subprogram and otherwise has the value false.

PRODUCT

Purpose To get the product of all the elements of an array.

Class Transformational function.

Syntax
```
FUNCTION PRODUCT(ARRAY, DIM [, MASK])
FUNCTION PRODUCT(ARRAY [, MASK])
LOGICAL MASK
INTEGER DIM
```

Description ARRAY is an array of type integer, real or complex. DIM is a scalar integer in the range $1 \leq \text{DIM} \leq n$ where n is the rank of ARRAY. MASK is a logical array that is conformable to ARRAY. The result will have the same attributes as ARRAY except that the rank will different. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank n - 1. If ARRAY has rank one and DIM is present then the given value of DIM is ignored and defaults to one.

Return value Case (i): The result of PRODUCT(ARRAY) has a value equal to the product of all the elements of ARRAY. It has the value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to the product of the elements of ARRAY corresponding to the elements of MASK that are true. It has the value one if there are no true elements.

Case (iii): The result of PRODUCT(ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to the product (using the mask if present) of the elements of ARRAY along the given dimension.

Example If A = $\begin{bmatrix}1,2,3\\4,5,6\end{bmatrix}$ then PRODUCT(A, DIM = 1) gives [4, 10, 18] and PRODUCT(A, DIM = 2) gives [6, 120].

RADIX

Purpose To get the base used internally for a given type and kind type.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION RADIX(X)
```

Description X is of type integer or real, a scalar or an array.

Return value The result is the scalar integer value of the model used by the compiler to represent values of the given type and kind type.

RANDOM_NUMBER

Purpose To get a pseudorandom number or an array of pseudorandom numbers in a uniform distribution over the range $0 \leq x < 1$.

Class Subroutine.

Syntax
```
SUBROUTINE RANDOM_NUMBER(HARVEST)
REAL,INTENT(OUT)::HARVEST
```

Description HARVEST may be a scalar which receives a single pseudorandom numbers or it may be an array to receive a sequence of pseudorandom numbers. The seed used by the random number generator can be set and interrogated using the subroutine RANDOM_SEED.

RANDOM_SEED

Purpose To restart or interrogate the pseudorandom number generator used by RANDOM_NUMBER.

Class Subroutine.

Syntax
```
SUBROUTINE RANDOM_SEED([SIZE, PUT, GET])
INTEGER, INTENT(OUT) SIZE
```

```
INTEGER, INTENT(IN)  PUT
INTEGER, INTENT(OUT) GET
```

Description RANDOM_SEED must have either no arguments or exactly one argument. SIZE is a scalar integer giving the number of integers *N* that are used by the random number generator to represent the seed. PUT is a one-dimensional array of size $\geq N$ that can be used to initialise the seed parameters. GET is a one-dimensional array of size $\geq N$ that can be used to get the current values of the seed parameters.

If RANDOM_SEED is called with no arguments then a default initial value is assigned to the seed.

RANGE

Purpose To get the range of the decimal exponent for integer or real numbers with a given kind type.

Class Inquiry function.

Syntax

```
INTEGER FUNCTION RANGE(X)
```

Description X is of type integer, real or complex. It may be a scalar or an array.

Return value The result is a scalar integer giving the difference between the largest (positive) exponent and the smallest (largest negative) exponent that is used by the compiler for numbers of the given type and kind type.

REAL

Purpose To convert to real type.

Class Elemental function.

Syntax

```
REAL FUNCTION REAL(A [,KIND]);  INTEGER KIND
```

Description A is of type INTEGER, REAL or COMPLEX.

If A is of type INTEGER or REAL, the result is REAL and kind type of the result is KIND if it is present otherwise it is the default real kind. If A is of type COMPLEX, the result is REAL and the kind type of the result is KIND otherwise it is the kind type of A (i.e. A is COMPLEX and the result is REAL with the same kind type value).

Return value If A is integer or real, the result is A.

If A is complex, the result is the real part A.

REPEAT

Purpose To concatenate several copies of a string.

Class Transformational function.

Syntax

```
CHARACTER (LEN=*) FUNCTION REPEAT(STRING, NCOPIES)
CHARACTER (LEN=*) STRING
INTEGER NCOPIES
```

Description NCOPIES is a non-negative value giving the number of times STRING is to appear in the result.

Return value The return is the resultant string.

RESHAPE

Purpose To construct an array of a specified shape from the elements of a given array.

Class Transformational function.

Syntax

```
FUNCTION RESHAPE(SOURCE, SHAPE [, PAD, ORDER])
INTEGER SHAPE,ORDER
```

Description SOURCE is an array of any type. If PAD is present it is also an array of the same type and type parameters as SOURCE which is used to fill in any values that are missing from the end of SOURCE. PAD is used repeatedly if necessary.

SHAPE is an one-dimensional integer array with *n* values where $0 < n < 8$. It gives the shape of the result (the number of elements along each dimension). If ORDER is present, it has the same shape as SHAPE and contains a permutation of the integers (1, 2, 3, . . ., *n*). The default is the natural order. ORDER is used to provide a permutation on the subscsript order of the result.

Return value The result is an array constructed from the elements of SOURCE and PAD using the shape given by SHAPE and ORDER.

Example RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value $\begin{bmatrix} 1,3,5 \\ 2,4,6 \end{bmatrix}$.

RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value $\begin{bmatrix} 1,2,3,4 \\ 5,6,0,0 \end{bmatrix}$.

RRSPACING

Purpose To get the reciprocal of the relative spacing of model numbers near the argument value.

Class Elemental function.

Syntax
```
REAL FUNCTION RRSPACING(X)
REAL X
```

Description See the ISO standard.

SCALE

Purpose To provide a model-based scaling of a real value.

Class Elemental function.

Syntax
```
REAL FUNCTION SCALE (X, I)
REAL X
INTEGER I
```

Return value The result is $X \times b^I$ where b is the base that is used by the compiler to model real numbers.

SCAN

Purpose To scan a string for any one of the characters in a set of characters.

Class Elemental function.

Syntax
```
INTEGER FUNCTION SCAN(STRING, SET [, BACK])
CHARACTER (LEN=*) STRING, SET
LOGICAL BACK
```

Description If BACK is omitted, this function scans STRING for the first ocurrence of any of the characters in SET. If BACK is set to true, scanning starts at the end of STRING.

Return value If a match is found, the result is an integer marking the position of the character in STRING starting at 1. If a match is not found or the length of STRING or SET is zero, the result is zero.

Example SCAN ('FORTRAN', 'TR') has the value 3.

SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

SELECTED_INT_KIND

Purpose To get the kind type parameter for integers with a given range.

Class Transformational function.

Syntax

```
INTEGER FUNCTION SELECTED_INT_KIND(R)
INTEGER R
```

Description The result is the kind type for integers *n* such that $-10^R < n < 10^R$.

Example SELECTED_INT_KIND(6) gives the kind type that may be used to represent values in the range $-999999 \leq n \leq 999999$.

SELECTED_REAL_KIND

Purpose To get the kind type parameter for real values with a given precision and range.

Class Transformational function.

Syntax

```
INTEGER FUNCTION SELECTED_REAL_KIND([P, R])
INTEGER P,R
```

Description SELECTED_REAL_KIND must have at least one argument. P is the required number of significant decimal digits (as returned by the intrinsic function PRECISION) and R is the required decimal exponent range (as returned by the intrinsic function RANGE).

Return value The result is the kind type parameter for real values with the given precision and/or range. A value of -1 is returned if P is out of range. A value of -2 is returned if R is out of range. A value of -3 is returned is both P and R are out of range.

SET_EXPONENT

Purpose To set the exponent of a real value relative to that used by model numbers.

Class Elemental function.

Syntax
```
REAL FUNCTION SET_EXPONENT(X, I)
REAL X
INTEGER I
```

Description See the ISO standard.

SHAPE

Purpose To get the shape of an array or a scalar.

Class Inquiry function.

Syntax
```
INTEGER FUNCTION SHAPE(SOURCE)
```

Description SOURCE is an array or scalar of any type.

Return value The result is a one-dimensional integer array whose size is equal to the rank of SOURCE and whose values represent the shape of SOURCE (the extent of each dimension in turn).

Example If A has dimension (2:5, -2:5) then SHAPE(A) gives (/ 4, 8 /). If B is a scalar then SHAPE(B) has rank one and zero size.

SIGN

Purpose To form a number from the absolute value of one number and the sign of another.

Class Elemental function.

Syntax
```
FUNCTION SIGN(A, B)
```

Description A can be integer or real. B and the result of the function have the same type as A.

Return value The result is formed by taking the absolute value of A and multiplying by -1 if B is negative. If B is zero the result may be unsafe.

SIN

Purpose To get the Sine of the argument.

Class Elemental function.

Syntax

```
REAL FUNCTION SIN(X);          REAL X
COMPLEX FUNCTION SIN(X);       COMPLEX X
```

Description If X is real it is an angle measured in radians. If X is complex its real part is in radians.

SINH

Purpose To get the Hyperbolic sine of the argument.

Class Elemental function.

Syntax

```
REAL FUNCTION SINH(X);         REAL X
```

Description The result has the same type as X.

SIZE

Purpose To get the extent of an array along a specified dimension or the total number of elements in the array.

Class Inquiry function.

Syntax

```
INTEGER FUNCTION SIZE(ARRAY [, DIM])
INTEGER DIM
```

Description ARRAY is an array of any type. If it is allocatable it must be allocated. If it is a pointer it must be associated. If it is an assumed size array then DIM must be present and have a value which is less than the rank of ARRAY.

If DIM is present it is a scalar in the range $1 \leq \text{DIM} \leq n$ where n is the rank of ARRAY.

Return value The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

SPACING

Purpose To get the absolute spacing of model numbers near the argument value.

Class Elemental function.

Syntax
```
REAL FUNCTION SPACING(X)
REAL X
```

Description See the ISO standard. If X is zero the result is the same as TINY(X).

SPREAD

Purpose To replicate an array by adding a dimension.

Class Transformational function.

Syntax
```
FUNCTION SPREAD(SOURCE, DIM, NCOPIES)
INTEGER DIM, NCOPIES
```

Description SOURCE is of any type, either an array or a scalar. The result has the same type as SOURCE and has rank $n + 1$ where n is the rank of SOURCE.

DIM is a scalar integer with an value in the range $1 \leq \text{DIM} \leq n$. NCOPIES is a scalar integer giving the number of time the original array is to be replicated.

Example If A = [6, 2, 3] then SPREAD(A, DIM = 2, NCOPIES = 4) gives $\begin{bmatrix} 6,6,6,6 \\ 2,2,2,2 \\ 3,3,3,3 \end{bmatrix}$

SQRT

Purpose To get the square root of the argument.

Class Elemental function.

Syntax
```
REAL    FUNCTION SQRT(X);  REAL    X
COMPLEX FUNCTION SQRT(X);  COMPLEX X
```

Description If X is real, it must not be negative.

If X is complex, the result is the principal value which has a real part that is non-negative. If the real part is zero, the imaginary part is non-negative.

SUM

Purpose To sum the elements of an array.

Class Transformational function.

Syntax

```
FUNCTION SUM(ARRAY, DIM [,MASK])
FUNCTION SUM(ARRAY [,MASK])
INTEGER DIM
```

Description ARRAY is an array of any type. The result is a scalar or array with the same type as elements of ARRAY. DIM is a scalar integer in the range $1 \leq DIM \leq n$ where n is the rank of ARRAY. MASK is an array of type logical which is conformable with ARRAY. Elements of MASK are set to true when corresonding elements of ARRAY are to included in the sum.

If DIM is present, summation is along the dimension specified by DIM resulting in an array of rank n - 1. If DIM is absent, the result is a scalar giving the sum of all the elements (for which MASK is true).

Example If A is $\begin{bmatrix} 1,2,3 \\ 4,5,6 \end{bmatrix}$ then SUM(A, DIM = 1) is [5, 7, 9] and SUM(A, DIM = 2) is [6, 15].

SYSTEM_CLOCK

Purpose To get integer data from a real-time clock.

Class Subroutine

Syntax

```
SUBROUTINE SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])
INTEGER,INTENT(OUT)::COUNT,COUNT_RATE,COUNT_MAX
```

Description COUNT is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count.

COUNT_RATE is the number of processor clock counts per second.

COUNT_MAX the maximum value that COUNT can have.

TAN

Purpose To get the Tangent of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION TAN(X);          REAL X
```

Description If X is real it is an angle measured in radians.

TANH

Purpose To get the Hyperbolic tangent of the argument.

Class Elemental function.

Syntax
```
REAL FUNCTION TANH(X);         REAL X
```

Description The result has the same type as X.

TINY

Purpose To get the smallest positive number of the model representing numbers of the same type and kind type parameter as the argument.

Class Inquiry function

Syntax
```
REAL FUNCTION TINY(X);         REAL X
```

Description X is a scalar or an array. The result is a scalar of the same type and kiind type.

TRANSFER

Purpose To change the type and/or kind type of a scalar or array.

Class Transformational function.

Syntax
```
FUNCTION TRANSFER (SOURCE, MOLD [, SIZE])
INTEGER SIZE
```

Description This function returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

SOURCE and MOLD may be of any type and may be scalar or array valued. SIZE is a scalar giving the size of the result. The result is of the same type and type parameters as MOLD.

Case (I): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is array valued and SIZE is absent, the result is array valued and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is array valued of rank one and size SIZE.

Return value If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is undefined. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) shall be the value of E.

TRANSPOSE

Purpose To transpose an array of rank two.

Class Transformational function.

Syntax

```
FUNCTION TRANSPOSE(MATRIX)
```

Description MATRIX is an array of any type and has rank two.

Return value The result is an array of the same type and type parameters as MATRIX and with rank two and shape (*n*, *m*) where (*m*, *n*) is the shape of MATRIX. Element (*i*, *j*) of the result has the value MATRIX (*j*, *i*).

TRIM

Purpose To remove trailing spaces from a string.

Class Transformational function.

Syntax

```
CHARACTER (LEN=*) FUNCTION TRIM(STRING)
CHARACTER (LEN=*) STRING
```

Description The result has the same kind type as STRING and a length that is the length of STRING less the number of trailing spaces in STRING.

UBOUND

Purpose To get the upper bound or upper bounds for the indexes of an array.

Class Inquiry function.

Syntax
```
FUNCTION UBOUND (ARRAY [, DIM])
INTEGER DIM
```

Description ARRAY is an array of any type. If it is a pointer it must be associated. If it is an allocatable array, it must be allocated. If it is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.

DIM is a scalar with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

The result is a scalar integer or a one-dimensional integer array (of default type). It is scalar if DIM is present, otherwise, the result is an array of size n.

Return value

Case (i): For an array section or for an array expression, other than a whole array or array structure component, UBOUND(ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the upper bound for subscript DIM of ARRAY. If dimension DIM has zero size the result is zero.

Case (ii): UBOUND(ARRAY) has a value whose ith component is equal to UBOUND(ARRAY, i), for $i = 1, 2, ..., n$.

Example If A is declared by the statement REAL A(2:3, 7:10) then UBOUND(A) is [3, 10] and UBOUND(A, DIM = 2) is 10.

UNPACK

Purpose Unpack a one-dimensional array under the control of a mask.

Class Transformational function.

Syntax
```
FUNCTION UNPACK(VECTOR, MASK, FIELD)
LOGICAL MASK
```

Description VECTOR is a one-dimensional array of any type. Its size is at least t where t is the number of true elements in MASK. MASK is a logical array. FIELD has the same

type and type parameters as VECTOR and is conformable with MASK.

The result is an array with the same type and type parameters as VECTOR and the same shape as MASK.

Return value Elements of VECTOR are unpacked into the result at positions where MASK is true. Unpacking starts along the first dimension (index) of MASK. When an element of MASK is false, FIELD is used if it is a scalar otherwise the corresponding element of FIELD is used.

Example If VECTOR = [2, 3, 4, 5, 6], MASK = $\begin{bmatrix}1,0,1\\0,1,0\\1,0,1\end{bmatrix}$ and FIELD = $\begin{bmatrix}7,7,7\\8,8,8\\9,9,9\end{bmatrix}$, then the result is $\begin{bmatrix}2,7,5\\8,4,8\\3,9,6\end{bmatrix}$.

VERIFY

Purpose To get the position of the first character in a string that does not appear in a given set of characters.

Class Elemental function.

Syntax

```
INTEGER FUNCTION VERIFY(STRING, SET [, BACK])
CHARACTER (LEN=*) STRING,SET
LOGICAL BACK
```

Description SET is of type character with the same kind type parameter as STRING. BACK is a scalar of type logical. The result is a scalar of default integer type.

Return value

Case (i): If BACK is absent or has the value false and if STRING contains at least one character that is not in SET, the value of the result is the position of the first character of STRING that is not in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is not in SET, the value of the result is the position of the rightmost character of STRING that is not in SET.

Case (iii): The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

Example VERIFY ('ABBA', 'A') has the value 2.

VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

VERIFY ('ABBA', 'AB') has the value 0.

24.

Overview of the Salford run-time library

This chapter contains outline information about the Win32 routines that are available in the Salford run-time library. Further information is available in the next chapter and in the on-line Help system. The routines below are arranged in functional groups with the given headings. Within the groups the routines are arranged in alphabetical order. Routines that are marked with a mouse symbol are only available under Win32. Other routines are available on all platforms: DOS, Win16 and Win32.

The Salford DBOS library of DOS related routines is described in the *FTN95 User's Supplement*. This supplement is provided in HTML form and can be found on the *Salford Tools CD*.

Index

Bit handling

The routines in this section provide for bit packed logical arrays. The routines are compiled in-line using the Intel bit manipulation instructions. The result executes about as fast as a reference to a logical array, but the data is stored 8 or 16 or 32 times more efficiently. The bit array may be held in an array (or even a simple variable) of any type (usually INTEGER (KIND=2)). These routines do not check that their arguments are in range, even in CHECK mode. Bits are numbered from 0. Bit 0 is the least significant bit of the first word of the array.

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

The routines in this section provide various facilities for manipulating objects of Fortran CHARACTER type.

ALLOCSTR@	Allocatew 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@	Fills 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@	Gets a string that was stored using ALLOCSTR@
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

The routines in this section allow the programmer to access the program name and command line arguments with which the program was invoked. If the program was run with /LGO or with the RUN77 utility under DOS, the command line arguments which are returned are those which follow the /PARAMS option (if one appears on the command line).

CMNAM	Reads a token from the command line.
CMNAM@	Reads a token from the command line.
CMNAMR	Resets the command line.
CMNARGS@	Gets the number of command line arguments.
CMNUM@	Gets the next command line argument as an integer.
CMPROGNM@	Gets the program name.
COMMAND_LINE	Reads the whole command line.
GET_PROGRAM_NAME@	Returns the name of the current program.

Console input and output

Routines that are nominally for screen and keyboard are strictly for standard output and standard input. As such, operaring system I/O redirection will work for these routines.

COU@	Outputs text to the screen with a new line.
COUA@	Outputs text to the screen without a new line.
ERRCOU@	Outputs text to the standard error device.
ERRCOUA@	Outputs text to the standard error device.
ERRNEWLINE@	Writes an newline to the standard error device.
ERRSOU@	Outputs text to the standard error device.
ERRSOUA@	Outputs text to the standard error device.
NEWLINE@	Writes a carriage return/linefeed to the screen (standard output).
PRINT_BYTES@	Writes a sequence of hexadecimal values.
PRINT_BYTES_R@	Writes 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(3) decimal number without a new line.
PRINT_R4@	Prints an REAL(1) decimal number without a new line.
PRINT_R8@	Prints an REAL(2) decimal number without a new line.
SOU@	Outputs text with a new line, omitting any trailing blanks.
SOUA@	Outputs text without a new line, omitting any trailing blanks.

Data sorting

The routines in this section provide facilities for sorting arrays of various types. The routines described use a quicksort algorithm, and perform well for data that is originally randomly ordered. Note, however, that these routines are not stable in the strict sense. That is, equal keys do not necessarily maintain their order relative to each other.

CHSORT@	Sorts an array of characters.
DSORT@	Sorts a REAL(KIND=2) array.
ISORT@	Sorts an integer array.
RSORT@	Sorts a REAL(KIND=1) array.

Error and exception handling

The routines described in this section fall into two main categories:

- Those which allow interpretation of error codes returned by other routines. Routines which fall into this category include ERR77 and DOS_ERROR_MESSAGE@. Where a routine returns an error code it is of course always good practice to check it for an acceptable value (usually zero).
- Those which allow control over the action taken in the event of a software-generated exception (such as an underflow, or a critical error). Further information is given in chapter 22.

ACCESS_DETAILS@	Gets details of the access violation.
CLEAR_FLT_UNDERFLOW@	Clears a floating point underflow exception.
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@	Finds the address of the instruction that generated the exception.
FORTRAN_ERROR_MESSAGE@	Gets a Fortran error message.
GET_VIRTUAL_COMMON_INFO@	Gets virtual common block details.
JUMP@	Executes a non-local jump.
LABEL@	Sets a label for a non-local jump.
PERMIT_UNDERFLOW@	Switches off floating point underflow checking.
PRERR@	Prints 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@	Removes a user defined exception handler.
RUNERR@	Prints the run-time error corresponding to a given IOSTAT value.
SET_TRAP@	Traps a given event.
TRAP_EXCEPTION@	Installs a user defined exception handler.
UNDERFLOW_COUNT@	Gets the number of floating point underflows.

File manipulation

FTN95 offers a wide variety of file manipulation routines. *Routines in this section that use a file handle must obtain the handle by calling one of the associated file opening routines***.** The reading and writing routines use a buffer to improve the performance. These buffers are cleared when a file is closed or when a program terminates. As a result, a file that has been open for writing may give an error (e.g. full disk) as it is closed with CLOSEF@. In all cases, where a routine returns an error code, this may be interpreted by calling a routine such as DOSERR@.

ATTACH@	Sets the current directory.
CLOSEF@	Closes a file.
CLOSEFD@	Closes and delete a file.
CURDIR@	Gets the current directory.
CURRENT_DIR@	Obsolete routine. Use CURDIR@
DIRENT@	Obtains directory information.
EMPTY@	Clears a file for writing.
ERASE@	Deletes a file.

FEXISTS@	Searches for a file with a given path name or wildcard.
FILE_EXISTS@	Obsolete routine. Use FEXIST@ instead
FILE_SIZE@	Gets the size of a file in bytes.
FILE_TRUNCATE@	Truncates an open file at its current position.
FILEINFO@	Gets information about a specified file.
FILES@	Obtains directory information.
FLUSH_FORTRAN_HANDLE@	Flushes the Fortran low level buffer.
FPOS@	Repositions a file.
FPOS_EOF@	Moves the file pointer to end-of-file.
GET_FILES@	Gets a list of files in the current working directory.
GET_PATH@	Gets the fully qualified pathname.
MKDIR@	Creates a new DOS directory.
OPENF@	Opens a file with sharing attributes.
OPENR@	Opens a file for reading.
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.
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.
TRANSLATE_FORTRAN_HANDLE@	Changes from a Salford file handle to a Win32 file handle.
WILDCHECK@	Checks for the matching of a file name with a wild card.
WRITEF@	Writes binary data to a file.
WRITEFA@	Writes a line of data to an ASCII file.

In-line

The routines in this section are converted to in-line code, rather than procedure calls, and are therefore extremely efficient. They can often be used as a convenient alternative to resorting to assembler code. Those routines that are functions should be explicitly declared to be of the right type.

FILL@	Sets an array of N bytes to a particular value.
MATCH@	Compares two arrays of N bytes.
MOVE@	Copies an array of N bytes.
POP@	Pops a value off the system stack.
PUSH@	Pushes a value on the system stack.

Process control

CISSUE	Issues a DOS command.
EXIT	Terminates a program.
EXIT@	Terminates a program.
SLEEP@	Suspends program execution for a specified time interval.
START_PROCESS@	Creates a new process.

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.

Serial communications

GETTERMINATECOMMCHAR@	Gets the character that terminated the last call to READCOMMDEVICE@.
OPENCOMMDEVICE@	Opens a serial port for I/O.
READCOMMDEVICE@	Reads data from an open serial port.
SETCOMMTERMINATECHAR@	Sets the characters that may be used to terminate a call to READCOMMDEVICE@.
SETECHOONREADCOMM@	Sets the communication port to echo back to the sending device.

WRITECOMMDEVICE@	Writes a string to a serial port.

Storage management

The routines described in this section relate to a virtual storage heap.

Since these routines work with addresses, the storage acquired by these routines must be manipulated by the 'core' intrinsics. As with any storage heap, it is important to avoid excessive fragmentation. This can be achieved by a variety of strategies, such as allocating blocks of fixed size, or deallocating all allocated storage at once, so that no 'holes' are created. It is not necessary to return allocated storage before a program terminates - this is done automatically.

Under Win32 there are two 100Mb heaps:

a) A Fortran heap used by GET_STORAGE@, RETURN_STORAGE@ and SHRINK_STORAGE@. This is fully virtual. Pages are provided by the run-time system in order to fill program page demands. The program may fail if it uses too much of the address space allocated or if the physical resources of the system are not sufficient to satisfy the program's demands.

b) A C/C++ heap used by **malloc**, **new** etc. in order to provide physical (committed) pages. Memory allocated from this heap is guaranteed to be available.

Memory allocated by GET_STORAGE@ should not be returned using **free** or **delete**. Similarly, memory allocated using **malloc** or **new** should not be returned using RETURN_STORAGE@.

GET_STORAGE@	Gets a block of storage of size N bytes from the storage heap.
RETURN_STORAGE@	Returns a block of storage.
SHRINK_STORAGE@	Shrinks a block of storage.

System information

DOSPARAM@	Gets an environment variable.
GET_CURRENT_FORTRAN_IO@	Accesses the state of the current Fortran I/O unit.
GET_CURRENT_FORTRAN_UNIT@	Gets the unit number for the current I/O operation.
GETENV@	Gets an environment variable.

Time and date

CLOCK@	Gets a time in seconds.
CONVDATE@	Gets the date in numeric form.
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@	Obtains the CPU time accurate to 1 microsecond.
SECONDS_SINCE_1980@	Gets the number of seconds from a fixed date.
TIME@	Gets the time in the format HH:MM:SS.
TODATE@	Converts a given time to a date in the form MM/DD/YY.
TOEDATE@	Converts a given time to a date in the form DD/MM/YY.
TOFDATE@	Returns the date in text form.
TOTIME@	Returns the time in the form HH:MM:SS.

25.

Salford run-time library

ACCESS_DETAILS@[1]

Purpose Get the details of the access violation.

Syntax
```
SUBROUTINE ACCESS_DETAILS@(ADDRESS, MODE, IC)
INTEGER (KIND=3) ADDRESS
LOGICAL (KIND=3) MODE
INTEGER (KIND=2) IC
```

Description This subroutine is used after an access violation has accured to ascertain the address that was being accessed when the exception occured. If this function is successful, *address* contains the address that was being accessed. *mode* is set to TRUE if the instruction was attempting to read from the address, FALSE if the instruction was attempting to write to the address.

ic is set to 0 on success, 1 on failure.

ALLOCSTR@

Purpose To allocate dynamic storage and copy a string.

Syntax
```
INTEGER (KIND=3) FUNCTION ALLOCSTR@(STRING)
CHARACTER (LEN=*) STRING
```

Description ALLOCSTR@ copies STRING with trailing spaces removed and terminated by a null (i.e. a C-format string), into a dynamic storage space which it allocates. The string can

[1] Routines marked with a mouse symbol are Win32 only.

be retrieved using the routine GETSTR@.

Return value The return value of the function is the address of the storage used.

Example See GETSTR@.

APPEND_STRING@

Purpose To add a string to the end of a line.

Syntax
```
SUBROUTINE APPEND_STRING@(LINE,ADDITION)
CHARACTER (LEN=*) LINE,ADDITION
```

Description This routine adds the string ADDITION to the end of string LINE after removing trailing spaces from LINE. This can be used to build up complex strings without the need to do many substring calculations.

Example
```
PROGRAM SONG
 CHARACTER (LEN=80)::LINE
 CHARACTER (LEN=20)::SAYINT
 INTEGER::NO_GREEN_BOTTLES
 LINE='THERE ARE'
 READ *,NO_GREEN_BOTTLES
 IF(NO_GREEN_BOTTLES == 0)THEN
  CALL APPEND_STRING@(LINE,' NO')
 ELSE
  CALL APPEND_STRING@(LINE,' '//SAYINT(NO_GREEN_BOTTLES))
 ENDIF
 CALL APPEND_STRING@(LINE,' STANDING ON A WALL')
 CALL SOU@(LINE)
END PROGRAM SONG
```

ATTACH@

Purpose To set the current directory.

Syntax
```
SUBROUTINE ATTACH@(PATH,ERROR_CODE)
CHARACTER (LEN=*) PATH
INTEGER (KIND=2) ERROR_CODE
```

Description PATH should be the pathname of a directory (e.g. C:\PROJECT). ATTACH@

makes this the current directory, switching disks if necessary. ERROR_CODE is returned with a non-zero system error code if it fails.

Example

```
CHARACTER (LEN=50)::PATH
CALL COUA@('Where do you want to be? ')
READ(*,'(A)')PATH
CALL ATTACH@(PATH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
CALL COU@('OK - that is where you are! ')
. . .
```

CENTRE@

Purpose To position a string in the centre of a field.

Syntax

```
CHARACTER (LEN=*) FUNCTION CENTRE@(STRING,IW)
CHARACTER (LEN=*) STRING
INTEGER (KIND=2) IW
```

Return value CENTRE@ returns STRING after padding with blanks on the left so that the non-blank part is centred in a field of IW characters. This is very useful for titles.

Example

```
CHARACTER (LEN=80) CENTRE@
PRINT *,CENTRE@('FINAL RESULTS',80)
```

CHAR_FILL@

Purpose To fill a string with a particular character.

Syntax

```
SUBROUTINE CHAR_FILL@(STRING,FILL)
CHARACTER (LEN=*) STRING
CHARACTER (LEN=1) FILL
```

Description This routine fills the string in STRING with the character FILL up to the full length of STRING.

Notes For new code, standard Fortran 95 intrinsic function REPEAT should be used instead.

CHSEEK@

Purpose To look for a given string in an ordered array.

Syntax

```
SUBROUTINE CHSEEK@(ITEM,LIST,N,IRES)
CHARACTER (LEN=*) ITEM,LIST(N)
INTEGER (KIND=3) N,IRES
```

Description Seeks the string ITEM in the sorted array LIST using a binary chop. Returns the position in IRES or 0 if not found. Note that the LIST array must be sorted in ascending dictionary order.

Example

```
PROGRAM FOOD
 CHARACTER(LEN=10)::FOODS(5)
 CHARACTER(LEN=12)::MEAL
 INTEGER::K
 DATA FOODS/'BUTTER','EGGS','FISH','MUTTON','SUGAR'/
 DO
   READ (*,'(A)') MEAL
   CALL UPCASE@(MEAL)
   CALL CHSEEK@(MEAL,FOODS,5,K)
   SELECT CASE(K)
     CASE(0)
       PRINT *,'this is not a food I know about'
       EXIT
     CASE(1)
       PRINT *,'spread it'
     CASE(2)
       PRINT *,'boil it'
     CASE(3)
       PRINT *,'fry it'
     CASE(4)
       PRINT *,'stew it'
     CASE(5)
       PRINT *,'put it in coffee'
   END SELECT
 END DO
END PROGRAM FOOD
```

CHSORT@

Purpose To sort an array of characters.

Syntax

```
SUBROUTINE CHSORT@(A,CHS,N)
CHARACTER (LEN=*) CHS(N)
INTEGER (KIND=3) A(N),N
```

Description CHSORT@ sorts the character array CHS by setting pointers from 1 to N in the array A. After sorting, A(1) contains a pointer to the "first" element of CHS, A(2) to the "second", and so on.

Example

```
PROGRAM SORT
  CHARACTER (LEN=20)::PUPILS(100)
  INTEGER (KIND=3)::I,IP(100)
  DO I=1,100
    READ(5,'(A)') PUPILS(I)
  END DO
  CALL CHSORT@(IP,PUPILS,100)
  PRINT *,'Sorted list of pupils:-'
  DO I=1,100
    PRINT *,PUPILS(IP(I))
  END DO
END PROGRAM SORT
```

CISSUE

Purpose To issue a system command.

Syntax

```
SUBROUTINE CISSUE(A,IFAIL)
CHARACTER (LEN=*) A
INTEGER (KIND=2) IFAIL
```

Description Issues the command stored as a character string in A. IFAIL is returned as one of the following:

IFAIL	Meaning
0	Successful invocation of a command processor to execute the command
1	A command processor could not be invoked

Notes The value of IFAIL refers to the success or failure of invoking the MS-DOS command processor COMMAND.COM. Unfortunately, MS-DOS does not provide a mechanism whereby the success or failure of the invocation of the particular command can be reported back to the caller. So, for example, if you get a system error such as "Not found" for the command, IFAIL will be returned as zero.

It is not possible to issue a command which itself uses DBOS (e.g. run another FTN95 program). Also the use of CISSUE to start TSR programs should be avoided since this can fragment memory.

Example

```
CALL COU@('the contents of this Directory are:-')
CALL CISSUE('DIR',K)
IF(K /= 0) CALL COU@('DIR failed for some reason')
. . .
```

CLEAR_BIT@

Purpose To clear the N'th bit of an array.

Syntax

```
SUBROUTINE CLEAR_BIT@(IA,N)
INTEGER (KIND=2) IA(*),N
```

Description Clears the N'th bit of the array IA. N can be INTEGER (KIND=1), (KIND=2) or (KIND=4) and IA can be of any datatype.

CLEAR_FLT_UNDERFLOW@

Purpose Clear a floating point underflow exception.

Syntax

```
SUBROUTINE CLEAR_FLT_UNDERFLOW@
```

Description Decode the instruction that caused the floating point underflow and clear the floating point underflow from the machine state.

CLOCK@

Purpose To get a time in seconds.

Syntax
```
SUBROUTINE CLOCK@(R)
REAL (KIND=1) R
```

Description This routine is usually used to time a process as shown in the example below.

Notes It should not be used to time processes under DESQview as it returns elapsed time, which is *not* CPU time, if multiple windows are in use.

See also DCLOCK@, HIGH_RES_CLOCK@, SECONDS_SINCE_1980@.

Example
```
        CALL CLOCK@(START)
        . . .
 !      some calculation
        . . .
        CALL CLOCK@(FINISH)
        PRINT *,'elapsed time used = ',FINISH-START
```

CLOSEF@

Purpose To close a file.

Syntax
```
SUBROUTINE CLOSEF@(HANDLE,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description CLOSEF@ closes a file opened by OPENR@, OPENW@ or OPENRW@. ERROR_CODE is returned with a non-zero system error code if it fails.

Example See OPRNRW@

CLOSEFD@

Purpose To close and delete a file.

Syntax
```
SUBROUTINE CLOSEFD@(HANDLE,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description CLOSEFD@ is the same as CLOSEF@ but CLOSEFD@ also deletes the file from

the disc. This is useful for temporary files.

CMNAM

Purpose To read a token from the command line.

Syntax `CHARACTER (LEN=*) FUNCTION CMNAM()`

Description This function is useful for programs which are to be used as commands with arguments on the command line. Arguments (or "tokens") are separated by spaces (not slashes).

Return value CMNAM reads a token and returns it as the character result. Spaces are returned when tokens are exhausted.

Example

```
!   This program operates like the MS-DOS COPY command.
!   Eg. store the program in COPY.FOR and type:
!   FTN95 COPY /LGO /PARAMS <first file> <second file>
    PROGRAM COPY_FILE
      CHARACTER (LEN=20)::CMNAM,FILE2
      CHARACTER (LEN=128)::DATA
      INTEGER (KIND=2)::KEY
      LOGICAL::L
      OPEN(UNIT=3,FILE=CMNAM(),STATUS='READONLY',ERR=200)
      FILE2=CMNAM()
      INQUIRE(FILE=FILE2,EXIST=L)
      IF (L) THEN
        PRINT *,'File to be written already exists&
                 & OK to overwrite? (y/n)')
        CALL GET_KEY@(KEY)
        IF((KEY /= 89).AND.(KEY /= 121)) STOP
        PRINT*,'y'
      ENDIF
      OPEN(UNIT=4,FILE=FILE2,STATUS='MODIFY',ERR=210)
!     Copy file one line at a time
      DO
        READ(3,'(A)',END=220) DATA
        WRITE(4,'(A)') DATA
      END DO
200   PRINT*,'File to be read does not exist'
      STOP
210   PRINT*,'File to be written can not be created'
220 END PROGRAM COPY_FILE
```

CMNAM@

Purpose To read a token from the command line.

Syntax
```
CHARACTER (LEN=*) FUNCTION CMNAM@()
```

Description This routine performs like CMNAM except that a token terminates at a slash. Slashes are converted into minus (-) characters and start the next token. For example, /ALPHA/BETA would generate two tokens: "-ALPHA" and "-BETA".

CMNAMR

Purpose To reset the command line.

Syntax
```
SUBROUTINE CMNAMR
```

Description CMNAMR resets the command line so that it can be read again with CMNAM etc.. CMNAMR has no arguments.

CMNARGS@

Purpose To get the number of command line arguments.

Syntax
```
INTEGER (KIND=2) FUNCTION CMNARGS@()
```

Return value CMNARGS@ returns the number of arguments on the command line (excluding the invocation name of the program).

CMNUM@

Purpose To get the next command line argument as an integer.

Syntax
```
SUBROUTINE CMNUM@(N,*)
INTEGER (KIND=3) N
```

Description This routine interprets the next command line argument as an integer if possible. The alternate return is taken if the next token is not a valid integer, and the next call of CMNAM@ will return the token which caused the error.

CMPROGNM@

Purpose To get the program name.

Syntax
```
CHARACTER (LEN=*) FUNCTION CMPROGNM@()
```

Return value CMPROGNM@ returns the name by which the program was invoked from the command line.

CNUM

Purpose To convert an integer to character form.

Syntax
```
CHARACTER (LEN=*) FUNCTION CNUM(J)
INTEGER (KIND=3) J
```

Description Converts the INTEGER (KIND=3) number J to characters, left-justified with sign if negative.

Example
```
!       routine to open a file of name FREDnnnn
!       where nnnn is a 4-digit integer
        SUBROUTINE FREDOPEN(K)
          INTEGER (KIND=3)::K
          CHARACTER (LEN=8)::FRED,CNUM
!       note trick to get leading zeros
          FRED(4:8) = CNUM(K+10000)
          FRED(1:4) = 'FRED'
          OPEN (FILE=FRED,UNIT=1)
        END SUBROUTINE FREDOPEN
```

COMMAND_LINE

Purpose To read the whole command line.

Syntax
```
SUBROUTINE COMMAND_LINE(C)
CHARACTER (LEN=*) C
```

Description This routine returns the whole command line in C (excluding the program name).

COMPRESS@

Purpose To compress a string by using tabs.

Syntax
```
SUBROUTINE COMPRESS@(LINE,L)
CHARACTER (LEN=*) LINE
INTEGER (KIND=2) L
```

Description COMPRESS@ replaces multiple blanks where possible in a line with tabs to column positions which are multiples of eight. The new length of the line is returned in L. The tabbing scheme is that used by DOS, so the resulting line can be written to a DOS file.

CONVDATE@

Purpose To get the date in numeric form.

Syntax
```
SUBROUTINE CONVDATE@(SECS,IDW,IDAY,IMONTH,IYEAR)
INTEGER (KIND=3) SECS
INTEGER (KIND=2) IDW,IDAY,IMONTH,IYEAR
```

Description Converts SECS into a day of the week, IDW (0 = Sunday) and the day, month and year.

COU@

Purpose To output text to the screen with a new line.

Syntax
```
SUBROUTINE COU@(A)
CHARACTER (LEN=*) A
```

Description COU@ takes the message length (which must be less than 1024 characters) from its character argument A.

See also COUA@, SOU@, SOUA@.

Example
```
CALL COU@('message to screen')
```

COUA@

Purpose To output text to the screen without a new line.

Syntax
```
SUBROUTINE COUA@(A)
CHARACTER (LEN=*) A
```

Description COUA@ takes the message length (which must be less than 1024 characters) from its character argument A. This is useful as a prompt or as part of a sequence of calls which build up a line on the screen.

See also COU@, SOU@, SOUA@.

Example
```
CALL COUA@('enter number of samples: ')
READ *,N
. . .
```

CPU_CLOCK@

Purpose To get the number of CPU clocks since start up.

Syntax
```
REAL (KIND=3) FUNCTION CPU_CLOCK@()
```

Description This routine is usually used to time a process as shown in the example below.

Notes CPU_CLOCK@ returns the value of the processor instruction 'rdtsc' which is only available with Pentium and Pentium Pro processors.

Under Windows 95 the accuracy of CPU_CLOCK@ is relatively poor because interrupts can not be disabled. It may not work under Windows NT.

The routine will have its own time penalty (of the order of 20 CPU clocks).

CURDIR@

Purpose To get the current directory.

Syntax
```
CHARACTER (LEN=*) FUNCTION CURDIR@()
```

Return value CURDIR@ returns the fully qualified pathname of the current directory.

Example

```
CHARACTER (LEN=50)::CURDIR@
PRINT *,'You are currently in ',CURDIR@()
```

CURRENT_DIR@

Purpose To get the current directory.

Syntax

```
SUBROUTINE CURRENT_DIR@(DIR,ERROR_CODE)
CHARACTER (LEN=*) DIR
INTEGER (KIND=2) ERROR_CODE
```

Description This routine is obsolete. Use CURDIR@ instead.

Return value Returns the name of the current working directory in DIR, or a non-zero error code ERROR_CODE if failed. ERROR_CODE is returned as ERANGE if the variable DIR is not of sufficient length. (ERANGE is defined in the include file *errno.ins*.)

DATE@

Purpose To get the date in the form MM/DD/YY or MM/DD/YYYY (American format).

Syntax

```
CHARACTER*8 FUNCTION DATE@()
CHARACTER*10 FUNCTION DATE@()
```

Description Use CHARACTER*8 for MM/DD/YY and CHARACTER*10 for MM/DD/YYYY where the year appears as four digits. If you set the environment variable DATEFAIL=YES, then a runtime error will occur when the 8-character form is used.

See also EDATE@, FDATE@.

Example

```
CHARACTER (LEN=8)::DATE@
PRINT *,'program run on ',DATE@()
```

DATE_TIME_SEED@

Purpose To select a new “seed” for the pseudo-random number generator function RANDOM.

Syntax

```
SUBROUTINE DATE_TIME_SEED@
```

Description This routine sets the seed for the random number generator to a value based on the current DATE/TIME. This routine is used to obtain a non repeatable sequence of pseudo-random numbers.

DCLOCK@

Purpose To get a time in seconds.

Syntax

```
SUBROUTINE DCLOCK@(R)
REAL (KIND=2) R
```

Description This routine is usually used to time a process as shown in the example for CLOCK@. This routine differs from CLOCK@ only in that its argument is REAL (KIND=2). Its main purpose is for use in conjunction with /DREAL, when all variables declared REAL (KIND=1) are actually compiled as REAL (KIND=2).

See also HIGH_RES_CLOCK@.

DIRENT@

Purpose To obtain directory information.

Syntax

```
 SUBROUTINE DIRENT@(PAT,ATTRIBUTE,RESULT,RESULT_ATTRIBUTE,&
&  RESULT_DATE,RESULT_TIME,FILE_SIZE,ERROR_CODE)
 CHARACTER (LEN=*) PAT,RESULT
 INTEGER (KIND=2) ATTRIBUTE,RESULT_ATTRIBUTE,RESULT_DATE, &
&  RESULT_TIME,ERROR_CODE
 INTEGER (KIND=3) FILE_SIZE
```

Description DIRENT@ returns directory information for files selected by PAT (e.g. A:*.FOR). That is, each call of the routine searches for a single file in the directory and with the extension implied by PAT. The attribute of the first file returned is selected by setting ATTRIBUTE to one of the following values:

0	Return a normal file
2	Return a hidden file
4	Return a system file

6	Return a volume name
16	Return a subdirectory

The name of the file that has been found is returned in RESULT. Other file information is returned in RESULT_ATTRIBUTE, RESULT_DATE, RESULT_TIME and FILE_SIZE. The file attributes are returned in DOS coded form using bits 0 to 5 of the result. The date and time are returned in DOS compressed format.

After the first call of the routine, ATTRIBUTE should be set to -1 in order to continue the search for another file with the same attribute as before. When no more files can be found, ERROR_CODE is returned with the corresponding system error code.

Notes A sequence of calls to DIRENT@ with a given PAT must not be interrupted by a call to DIRENT@ with a different PAT.

The FILES@ routine has a simpler interface and is usually preferred to this routine.

Example

```
PROGRAM DIRENT
  CHARACTER (LEN=20)::PAT,FILE
  INTEGER (KIND=2)::ATTR,DATE,TIME,EC
  INTEGER (KIND=3)::SIZE
  CALL COUA@('Input directory pattern:')
  READ '(A)',PAT
  EC=0
  DO WHILE(EC == 0)
    CALL DIRENT@(PAT,0,FILE,ATTR,DATE,TIME,SIZE,EC)
    IF (EC == 0) PRINT '(A,I6,I6,I6,I6)',FILE,ATTR,DATE,TIME,SIZE
    !Do not call DIRENT@(PAT2,...) from here
  END DO
END PROGRAM DIRENT
```

DOS_ERROR_MESSAGE@

Purpose To get a DOS error message.

Syntax

```
SUBROUTINE DOS_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
INTEGER (KIND=2) ERROR_CODE
CHARACTER (LEN=*) MESSAGE
```

Description Returns the DOS error string corresponding to the run time error number or IOSTAT value ERROR_CODE.

IOSTAT values of 10000 and above indicate that a file system error has occurred. The

system error code is determined by subtracting 10000. System documentation can then be used to decode the error. Alternatively, the error message can be obtained by passing the adjusted IOSTAT value to DOS_ERROR_MESSAGE@.

Example

```
CHARACTER (LEN=80)::MESSAGE
CALL OPENR@('DATA',IH,ERROR_CODE)
CALL DOS_ERROR_MESSAGE@(ERROR_CODE-10000,MESSAGE)
PRINT *,MESSAGE
```

DOSERR@

Purpose To print a DOS error message and exit when an error occurs.

Syntax

```
SUBROUTINE DOSERR@(ERROR_CODE)
INTEGER (KIND=2) ERROR_CODE
```

Description This routine does nothing if ERROR_CODE is zero, otherwise it prints the DOS error message corresponding to ERROR_CODE and exits from the program. It is typically used after system calls that use DOS and are normally expected to succeed.

Example:

```
CALL OPENR@('DATA',IH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
```

DOSPARAM@

Purpose To get a DOS environment parameter value.

Syntax

```
SUBROUTINE DOSPARAM@(PARAM,VALUE)
CHARACTER (LEN=*) PARAM,VALUE
```

Description This routine returns the value VALUE of a DOS parameter PARAM, which has been set using the DOS SET command. This can be very useful while creating environments in which programs are controlled by global information set up in batch files.

Example After the DOS command

```
SET FILENAME=FRED
```

has been executed, the following would open the file FRED:

```
CHARACTER (LEN=50)::FILE
CALL DOSPARAM@('FILENAME',FILE)
```

```
OPEN(FILE=FILE,UNIT=6)
```

DSORT@

Purpose To sort a REAL (KIND=2) array.

Syntax
```
SUBROUTINE DSORT@(A,D,N)
REAL (KIND=2) D(N)
INTEGER (KIND=3) A(N),N
```

Description DSORT@ sorts the REAL array D by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

EDATE@

Purpose To get the date in the form DD/MM/YY or DD/MM//YYYY (European format).

Syntax
```
CHARACTER (LEN=8) FUNCTION EDATE@()
CHARACTER (LEN=10) FUNCTION EDATE@()
```

Description Use CHARACTER (LEN=8) for DD/MM/YY and CHARACTER (LEN=10) for DD/MM/YYYY where the year appears as four digits. If you set the environment variable DATEFAIL=YES, then a runtime error will occur when the 8-character form is used.

See also DATE@, FDATE@.

Example
```
CHARACTER (LEN=8)::EDATE@
PRINT *,'program run on ',EDATE@()
```

EMPTY@

Purpose To clear a file for writing.

Syntax
```
SUBROUTINE EMPTY@(HANDLE,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description EMPTY@ clears the file open with file handle HANDLE (which must not be open for reading only). ERROR_CODE is returned as zero for success or with a system error

code if it fails.

ERASE@

Purpose To delete a file.

Syntax

```
SUBROUTINE ERASE@(FILE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) ERROR_CODE
```

Description ERASE@ deletes a file. The file name may be a local name, for example: FOO.TXT, or a fully qualified pathname, for example, C:\PROJECT\JUNK.TXT. ERROR_CODE is returned as zero for success or with the system error code.

Example

```
CALL ERASE@('USELESS.DAT',ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
```

ERR77

Purpose To print a DOS error message and terminate a program when an error occurs.

Syntax

```
SUBROUTINE ERR77(MESSAGE,ERROR_CODE)
CHARACTER (LEN=*) MESSAGE
INTEGER (KIND=2) ERROR_CODE
```

Description This routine has a null effect if ERROR_CODE is zero. Otherwise the string MESSAGE is printed, followed by the text of the DOS error indicated by ERROR_CODE. The program then terminates abnormally. This routine is normally used to test for DOS error following another system call, as in the example.

Example

```
CHARACTER (LEN=40)::FILE
READ(*,'(A)')FILE
CALL OPENR@(FILE,HANDLE,ERROR_CODE)
CALL ERR77(FILE,ERROR_CODE)
. . .
```

ERRCOU@

Purpose To output text to the standard error device.

Syntax
```
SUBROUTINE ERRCOU@(STRING)
CHARACTER (LEN=*) STRING
```

Description ERRCOU@ outputs text followed by a new line taking the message length from its character argument STRING.

ERRCOUA@

Purpose To output text to the standard error device.

Syntax
```
SUBROUTINE ERRCOUA@(STRING)
CHARACTER (LEN=*) STRING
```

Description ERRCOUA@ outputs text (without a new line) taking the message length from its character argument STRING. This is useful as a prompt or as part of a sequence of calls which build up a line on the screen.

ERRNEWLINE@

Purpose To write a newline to the standard error device.

Syntax
```
SUBROUTINE ERRNEWLINE@
```

ERROR@

Purpose To print a user defined error message and terminate a program.

Syntax
```
SUBROUTINE ERROR@(ERROR_MESSAGE)
CHARACTER (LEN=*) ERROR_MESSAGE
```

Description This routine generates a user defined error condition. If the program is running under the debugger, then the message will be displayed in the error window. Otherwise, the error is printed out and the EXIT routine is called with a code of 1 (exit to DOS).

Example

```
CALL ERROR@('Too many data points for PLOT program')
```

ERRSOU@

Purpose To output text to the standard error device.

Syntax

```
SUBROUTINE ERRSOU@(STRING)
CHARACTER (LEN=*) STRING
```

Description ERRSOU@ outputs text from the character argument STRING to standard error omitting any trailing blanks, and outputing a new line.

ERRSOUA@

Purpose To output text to the standard error device.

Syntax

```
SUBROUTINE ERRSOUA@(STRING)
CHARACTER (LEN=*) STRING
```

Description ERRSOUA@ outputs text from the character argument STRING to standard error omitting any trailing blanks. Does not output a new line.

EXCEPTION_ADDRESS@

Purpose To find the address of the intruction which generated the exception

Syntax

```
INTEGER (KIND=3) EXCEPTION_ADDRESS@
```

Description EXCEPTION_ADDRESS@ returns the address of the instruction that generated the exception event.

Return value Address of the instruction that generated the exception.

EXIT

Purpose To terminate a program.

Syntax
```
SUBROUTINE EXIT(ERROR_CODE)
INTEGER (KIND=2) ERROR_CODE
```

Description This routine terminates the program and returns to the operating system. If the error code is non zero the termination will be abnormal. Abnormal termination will interrupt the flow of a .BAT file (as if control break had been pressed).

EXIT@

Purpose To terminate a program.

Syntax
```
SUBROUTINE EXIT@(ERROR_CODE)
INTEGER (KIND=2) ERROR_CODE
```

Description EXIT@ is a synonym for EXIT.

FDATE@

Purpose To get the date in text form.

Syntax
```
CHARACTER (LEN=*) FUNCTION FDATE@()
```

Return value FDATE@ returns the date in the form:
Thursday February 11, 1988

See also DATE@, EDATE@.

Example
```
CHARACTER (LEN=20)::FDATE@
PRINT *,'Program run on ',FDATE@()
```

FEXISTS@

Purpose To search for a file with a given path name or wildcard.

Syntax
```
LOGICAL (KIND=3) FUNCTION FEXISTS@(PATH, ERROR_CODE)
CHARACTER (LEN=*) PATH
```

```
INTEGER (KIND=3) ERROR_CODE
```

Return value FEXISTS@ returns a logical value which is .TRUE. if the name supplied in PATH is that of a file which does exist, or is a wildcard which matches one file only. It returns .FALSE. if such a file does not exist, or if an error occurs in which case ERROR_CODE returns a non-zero system error code.

FILE_EXISTS@

Purpose To search for a file with a given path name or wildcard.

Syntax

```
LOGICAL (KIND=3) FUNCTION FILE_EXISTS@(PATH)
CHARACTER (LEN=*) PATH
```

Description This function is obsolete. Use FEXISTS@ instead.

FILE_EXISTS@ returns a logical value which is .TRUE. if the name supplied in PATH is that of a file which does exist, or is a wildcard which matches one file only. It returns .FALSE. if such a file does not exist, or if any kind of error occurs.

FILE_SIZE@

Purpose To get the size of FILE in bytes.

Syntax

```
SUBROUTINE FILE_SIZE@(FILE,SIZE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=3) SIZE
INTEGER (KIND=2) ERROR_CODE
```

FILE_TRUNCATE@

Purpose To truncate an open file at its current position.

Syntax

```
SUBROUTINE FILE_TRUNCATE@(HANDLE,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description This routine uses the handle of a file that has already been opened by OPENW@ or OPENRW@ and truncates the file at the current writing position. ERROR_CODE is returned as zero if the process is carried out successfully, otherwise ERROR_CODE

returns a system error code.

FILEINFO@

Purpose To get information about a specified file.

Syntax

```
 SUBROUTINE FILEINFO@(PATH,MODE,DEV,RDEV, &
& NLINK,SIZE,ATIME,MTIME,CTIME,ERROR_CODE)
 CHARACTER (LEN=*) PATH
 INTEGER (KIND=2) MODE,DEV,RDEV,NLINK,ERROR_CODE
 INTEGER (KIND=3) SIZE,ATIME,MTIME,CTIME
```

Description Returns information about the file specified by PATH. This routine can be used to return the size of a file in SIZE and the date and time that the file was last accessed in ATIME. The returned value of ATIME can then be supplied to TOTIME@, TOEDATE@ etc.. ERROR_CODE is returned as zero if the process is carried out successfully, otherwise ERROR_CODE returns a system error code.

MODE takes one of the following octal values:

DIREC	0040000	directory
ARCHIVE	0100000	regular
IRWXU	0000700	read, write, execute, owner
IRUSR	0000400	read, owner
IWUSR	0000200	write, owner
IXUSR	0000100	execute/search, owner
IRWXG	0000070	read, write, execute, group
IRGRP	0000040	read, group
IWGRP	0000020	write, group
IXGRP	0000010	execute/search, group
IRWXO	0000007	read, write, execute, other
IROTH	0000004	read, other
IWOTH	0000002	write, other
IXOTH	0000001	execute/search, other

Notes Arguments that do not appear in the above description are redundant in this operating system environment.

FILES@

Purpose To obtain directory information.

Syntax

```
 SUBROUTINE FILES@(PAT,N,NMAX,FILES,ATTR,DATE,TIME, &
&  FILE_SIZE)
 CHARACTER (LEN=*) PAT,FILES(NMAX)
 INTEGER (KIND=2) N,NMAX
 INTEGER (KIND=2) ATTR(NMAX),DATE(NMAX),TIME(NMAX)
 INTEGER (KIND=3) FILE_SIZE(NMAX)
```

Description Returns directory information for files selected by PAT (e.g. A:*.FOR). N is returned as the number of file names returned. If N is equal to NMAX there may be more matches which could not be returned. The remainder of the arrays return information about the files. Hidden files and directories are returned by this routine together with ordinary files. Such files may be distinguished by using the DOS file attribute returned in the ATTR array. The date and time are returned in the DOS compressed format. File attributes are combinations of the following hexadecimal flags:

NORMAL	Z'00'	normal
RDONLY	Z'01'	read only
HIDDEN	Z'02'	hidden
SYSTEM	Z'04'	system
LABEL	Z'08'	label
DIREC	Z'10'	directory
ARCHIVE	Z'20'	archive

Example See SET_FILE_ATTRIBUTE@.

FILL@

Purpose To set an array of N bytes to a particular value.

Syntax

```
SUBROUTINE FILL@(A,N,B)
INTEGER (KIND=3) A,B,N
```

Description This routine fills A (which may be of any type and is usually an array) with N bytes of value B. Thus if A is of type INTEGER*4 and N=4, each of the 4 bytes of A will be assigned to the value of B. B may be of any type but only the lowest byte is used.

FLUSH_FORTRAN_HANDLE@

Purpose To flush the Fortran low level buffer.

Syntax
```
SUBROUTINE FLUSH_FORTRAN_HANDLE@(HANDLE)
INTEGER (KIND=2) HANDLE
```

Description HANDLE is a handle returned by OPENRW@ or OPENF@. This routine flushes the output buffer to disk but keeps the file open.

FORTRAN_ERROR_MESSAGE@

Purpose To get the error message for a given run time error number or IOSTAT value.

Syntax
```
SUBROUTINE FORTRAN_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
INTEGER (KIND=2) ERROR_CODE
CHARACTER (LEN=*) MESSAGE
```

Description Returns the error message corresponding to the run time error number or IOSTAT value ERROR_CODE.

IOSTAT values of 10000 and above indicate that a file system error has occurred. The system error code is determined by subtracting 10000. System documentation can then be used to decode the error. Alternatively, the error message can be obtained by passing the unadjusted IOSTAT value to FORTRAN_ERROR_MESSAGE@.

Example
```
CHARACTER (LEN=80)::MESSAGE
INTEGER (KIND=2)::ERROR_CODE
OPEN(FILE='FRED',UNIT=6,IOSTAT=ERROR_CODE)
CALL FORTRAN_ERROR_MESSAGE@(ERROR_CODE,MESSAGE)
PRINT *,MESSAGE
```

FPOS@

Purpose To reposition a file.

Syntax
```
SUBROUTINE FPOS@(HANDLE,POSITION,NEW_POSITION,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
INTEGER (KIND=3) POSITION,NEW_POSITION
```

Description FPOS@ attempts to reposition an open file with the given HANDLE to the given POSITION. NEW_POSITION is returned as either the requested POSITION or as the position of end-of-file, whichever is less. ERROR_CODE is returned as zero or with a system error code if it fails.

Notes If the input value of POSITION is supplied as a constant and the compiler option /INTS is used, it will be necessary to force the length of the constant to 4 bytes.

FPOS_EOF@

Purpose To move the file pointer to end-of-file

Syntax
```
SUBROUTINE FPOS_EOF@(DESC,NEW_POSITION,ERROR_CODE)
INTEGER (KIND=2) DESC,ERROR_CODE
INTEGER (KIND=3) NEW_POSITION
```

Description Move the file pointer associated with the file open on DESC to end-of-file. NEW_POSITION is the new value of the file pointer.

GET_CURRENT_FORTRAN_IO@

Purpose To access the state of the current Fortran I/O unit.

Syntax
```
 SUBROUTINE GET_CURRENT_FORTRAN_IO@(UNIT,REC,RECL,STATUS, &
&  NBYTES)
 INTEGER (KIND=2) UNIT,STATUS
 INTEGER (KIND=3) REC,RECL,NBYTES
```

Description This routine supersedes GET_CURRENT_FORTRAN_UNIT@. This allows a Fortran device driver to access the state of the current unit. This is useful when a device driver is attached to multiple units. The current unit is returned in UNIT, the current record in REC and the current record length in RECL. NBYTES is filled with the number of bytes to read/write.
STATUS is filled thus:

Bit-0 set for FORMATTED, unset for UNFORMATTED I/O,
Bit-1 set for DIRECT, unset for SEQUENTIAL access.

GET_CURRENT_FORTRAN_UNIT@

Purpose To get the unit number for the current I/O operation.

Syntax
```
SUBROUTINE GET_CURRENT_FORTRAN_UNIT@(UNIT)
INTEGER (KIND=2) UNIT
```

Description This routine is useful in device drivers (using the DEVICE= I/O keyword) which are to be attached to more than one Fortran stream.

GET_FILES@

Purpose To get a list of files in the current working directory.

Syntax
```
 SUBROUTINE GET_FILES@(WILDCARD,FILES, &
&  MAXFILES,NFILES,ERROR_CODE)
 CHARACTER (LEN=*) WILDCARD,FILES(MAXFILES)
 INTEGER (KIND=2) MAXFILES,NFILES,ERROR_CODE
```

Description Returns a list of all the files in the current working directory which can be matched by WILDCARD.

Returns error code ERROR_CODE as ERANGE if FILES is not big enough, but the entries which are stored in FILES will be valid.

GET_KEY@

Purpose To get the next keycode.

Syntax
```
SUBROUTINE GET_KEY@(K)
INTEGER (KIND=2) K
```

Description This subroutine gets a keycode from the keyboard. If the buffer is not empty then a value is read and removed from the buffer. If the buffer is empty then the function waits for a keyboard input. The returned value of K is such that if the high byte is equal to 0 then the low byte is the ASCII value for the key pressed.

Under DBOS, if the high byte is equal to 1 then the low byte is the scan code for a Function key or an ALT key combination (the scan code is then K-256).

Under Win32, if the least significant bit of the high byte is set, then the low byte is the scan code for a Function key or a ALT key combination. The next three bits of the

high byte are set when respectively the ALT, CTRL and SHIFT keys are pressed.

See also GET_KEY_OR_YIELD@.

Example

```
CALL GET_KEY@(K)
IF(K == Z'13B')THEN
  PRINT *,'F1 Key pressed'
ELSEIF(K == Z'13C')THEN
  :
  :
ENDIF
```

GET_KEY_OR_YIELD@

Purpose To get the next keycode.

Syntax

```
SUBROUTINE GET_KEY_OR_YIELD@(KEY)
INTEGER (KIND=2) KEY
```

Description This routine returns a key typed on the keyboard in exactly the same way as GET_KEY@ except that it will yield control to other tasks (if any) when no keypress is pending. This routine is used in READ_EDITED_LINE@ and WREAD_EDITED_LINE@. Care should be taken to ensure that only one process is performing keyboard input with these routines at any one time - otherwise the characters will be shared randomly across several tasks!

GET_LAST_IO_ERROR@

Purpose To get the file system error number corresponding to the last IOSTAT value returned.

Syntax

```
INTEGER(KIND=3) GET_LAST_IO_ERROR@()
```

Description IOSTAT values of 10000 and above indicate that a file system error has occurred. The system error code is determined by subtracting 10000. GET_LAST_IO_ERROR@() gives the system error code for the last IOSTAT value returned (i.e. with 10000 subtracted). This value is used with DOS_ERROR_MESSAGE@. The unadjusted value is used with FORTRAN_ERROR_MESSAGE@.

GET_PATH@

Purpose To get the fully qualified pathname.

Syntax
```
SUBROUTINE GET_PATH@(HANDLE,RESULT,ERROR_CODE)
CHARACTER (LEN=*) RESULT
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description GET_PATH@ returns the pathname of the file open on file handle HANDLE. This works regardless of whether a local or global name was used when the file was originally opened. ERROR_CODE is returned as zero for success or it is returned as a system error code.

Example
```
CHARACTER (LEN=100)::FULL_PATH
CALL OPENR@('MYDATA',HANDLE,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
CALL GET_PATH@(HANDLE,FULL_PATH,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
PRINT *,FULL_PATH
. . .
```

GET_PROGRAM_NAME@

Purpose To return the name of the current program.

Syntax
```
SUBROUTINE GET_PROGRAM_NAME@(NAME)
CHARACTER (LEN=*) NAME
```

GET_STORAGE@

Purpose To get a block of storage of size N bytes from the storage heap.

Syntax
```
SUBROUTINE GET_STORAGE@(ADDR,N)
INTEGER (KIND=3) ADDR,N
```

Description ADDR is returned as the address of the first byte of the block. Under DBOS a returned value of -1 indicates that there is insufficient contiguous storage to create the block. Under Win32 a returned value of zero indicates that there is insufficient contiguous storage to create the block

Notes The heap used by GET_STORAGE@, RETURN_STORAGE@ and SHRINK_STORAGE@

is fully virtual. Pages are provided by the runtime system to satisfy the program's page demands. The program may fail if it uses too much of the address space allocated or if the physical resources on the system are not sufficient to satisfy the program's demands.

Memory allocated using GET_STORAGE@ should be returned using RETURN_STORAGE@. The C function **free** and the C++ operator **delete** can not be used for this purpose.

GET_VIRTUAL_COMMON_INFO@

Purpose Get virtual common block details.

Syntax

```
 SUBROUTINE GET_VIRTUAL_COMMON_INFO@(NAME,BASE, &
&      SIZE,COMMIT,AMOUT_COMMITTED)
 CHARACTER (LEN=*) NAME
 INTEGER (KIND=3) BASE,SIZE,COMMIT,AMOUT_COMMITTED
```

Description When a program is linked using the virtual common (VC) option, all uninitialised data (i.e. the BSS section) is removed from the executable and placed into virtual paged memory with default base address 0x20000000. Pages of memory are committed when necessary. GET_VIRTUAL_COMMON_INFO@ allows a user to determine how much memory is used and also provides other information.

The subroutine returns BASE, SIZE, COMMIT (amount of memory the system automatically commits) and AMOUNT_COMITTED (amount of memory already committed) for the allocated virtual common block.

Example

```
PROGRAM TVC1
  INTEGER (KIND=3)::BASE,SIZE,COMMIT,AMT_COMMIT
  CALL GET_VIRTUAL_COMMON_INFO@('TVC1.EXE',BASE, SIZE, COMMIT,AMT_COMMIT)
  PRINT*,' BASE = ', BASE
  PRINT*,' SIZE = ', SIZE
  PRINT*,' COMMIT = ', COMMIT
  PRINT*,' AMT_COMMIT = ', AMT_COMMIT
END PROGRAM TVC1
```

GETENV@

Purpose To get an environment variable.

Syntax
```
CHARACTER (LEN=*) FUNCTION GETENV@(VARIABLE)
CHARACTER (LEN=*) VARIABLE
```

Return value Returns the value of the specified environment variable.

GETSTR@

Purpose To get a string which was stored using ALLOCSTR@.

Syntax
```
CHARACTER (LEN=*) FUNCTION GETSTR@(PTR)
INTEGER (KIND=3) PTR
```

Description This function can be used for strings allocated with the ALLOCSTR@ routine. ALLOCSTR@ and GETSTR@ provide a simple way of storing and retrieving large amounts of character information for which a maximum possible length of each element is known, but where if all trailing spaces were stored the amount of memory required would be excessive. For example, lines of text destined for screen display could be stored in this way (usually a maximum of 80 characters, but often with much trailing space).

Another application of this routine is for C string entities passed to Fortran routines (see chapter 15).

Return value GETSTR@ returns the null-terminated string at address PTR as a Fortran CHARACTER entity, truncating or blank-padding as necessary.

Example
```
INTEGER (KIND=3)::ALLOCSTR@,PTR
CHARACTER (LEN=80):: GETSTR@
. . .
PTR=ALLOCSTR@(LINE)
. . .
OUTLIN=GETSTR@(PTR)
PRINT '(A)', OUTLIN
```

GETTERMINATECOMMCHAR@

Purpose To get the character that terminated the last call to READCOMMDEVICE@.

Syntax
```
CHARACTER GETTERMINATECOMMCHAR@(PORTNUM)
INTEGER (KIND=3) PORTNUM
```

Return value GETTERMINATECOMMCHAR@ returns the character that terminated the last READCOMMDEVICE@ call. This will be one of the characters set using SETCOMMTERMINATECHAR@. If the port is not open the function returns -1. If READCOMMDEVICE@ has not been called the default return is zero.

HIGH_RES_CLOCK@

Purpose To obtain the CPU time accurate to 1 microsecond.

Syntax
```
REAL (KIND=2) FUNCTION HIGH_RES_CLOCK@(ALIGN)
LOGICAL (KIND=2) ALIGN
```

Description The DBOS function returns the CPU time as seconds since midnight accurate to about 1 microsecond (although the cost of the function call is approximately 100 microseconds because it must call DBOS). To achieve this precision the system clock is reprogrammed in mode 2. This could in principle affect other software, although we are not aware of any problems. The clock remains programmed in mode 2 until the system is rebooted. If the ALIGN argument is set .TRUE., this function will not return until after the next clock tick to help to obtain consistent timings. Obviously the second of a pair of calls to HIGH_RES_CLOCK@ should have ALIGN set to .FALSE.. Although the function is defined as REAL (KIND=2) it actually returns an 80-bit (KIND=3) precision result.

The Win32 function uses the API function **QueryPerformanceCounter** to return a value in seconds relative to a fixed initial time. The argument ALIGN is not used.

Notes The DBOS function should not be used under Windows 95/98 because it will adversely affect the subsequent system performance.

See also DCLOCK@.

Example
```
REAL (KIND=2)::T1,T2,HIGH_RES_CLOCK@
T1=HIGH_RES_CLOCK@(.TRUE.)
CALL SOME_PROCESS
T2=HIGH_RES_CLOCK@(.FALSE.)
PRINT *,'Time required = ',T2-T1
```

ISORT@

Purpose To sort an integer array.

Syntax
```
SUBROUTINE ISORT@(A,IA,N)
INTEGER (KIND=3) IA(N)
INTEGER (KIND=3) A(N),N
```

Description ISORT@ sorts the integer array IA by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

JUMP@

Purpose To execute a non-local jump.

Syntax
```
SUBROUTINE JUMP@(LABEL)
COMPLEX (KIND=2) LABEL
```

Description This routine is described here since its most frequent use is in conjuction with SET_TRAP@. It takes a label generated by LABEL@ and jumps to the label. The label must exist in a still active routine, but this can be any distance down the call stack.

Example Consider a program designed to process several sets of data and to carry on even after errors had been diagnosed in earlier data sets:

```
      PROGRAM INPUT
          COMPLEX (KIND=2) RECOVER
        CALL LABEL@(RECOVER,*10)
10       DO
            CALL READ_DATA
          CALL PROCESS
          END DO

      CONTAINS

      SUBROUTINE ERROR(MESSAGE)
        CHARACTER*(*) MESSAGE
        CALL COU@(MESSAGE)
        CALL JUMP@(RECOVER)
      END SUBROUTINE ERROR

        END PROGRAM INPUT
```

Subroutine ERROR could be called from anywhere inside PROCESS (even many layers down inside subroutine calls) to return to label 10 ready to process the next data set. The use of JUMP@ obviates the need to provide an explicit error exit path back to the main program. It is the user's responsibility to ensure that the LABEL is still accessible when JUMP@ is called, i.e. that the routine which is called LABEL@ has not yet exited.

LABEL@

Purpose To set a label for a non-local jump.

Syntax
```
SUBROUTINE LABEL@(LABEL,*)
COMPLEX (KIND=2) LABEL
```

Description This routine makes a label for use with JUMP@.

Example See JUMP@.

LCASE@

Purpose To alter a character argument so that all letters become lower case.

Syntax
```
SUBROUTINE LCASE@(A)
CHARACTER (LEN=*) A
```

Example
```
PROGRAM TEST_CASE
  CHARACTER (LEN=10)::FRED
  FRED = 'ABC123'
  CALL LCASE@(FRED)
  IF(FRED /= 'abc123') PRINT *,'LCASE@ routine has failed'
END PROGRAM TEST_CASE
```

MATCH@

Purpose To compare two arrays of N bytes.

Syntax
```
LOGICAL (KIND=2) FUNCTION MATCH@(A,B,N)
INTEGER (KIND=2) A,B,N
```

Description This function compares N bytes of data for equality. The first two arguments can be of any type and are usually arrays. N can be INTEGER (KIND=1), (KIND=2) or (KIND=3).

Return value MATCH@ returns .TRUE. if the two arrays are identical, otherwise .FALSE..

MKDIR@

Purpose To create a new system directory.

Syntax
```
SUBROUTINE MKDIR@(DIR,ERROR_CODE)
CHARACTER (LEN=*) DIR
INTEGER (KIND=2) ERROR_CODE
```

Description The argument DIR can be either the local name of a directory, or the full path name. In either case, if the directory cannot be created for any reason, a non-zero system error code will be returned.

Example
```
CALL MKDIR@('C:\ACCOUNTS',IC)
CALL DOSERR@(IC)
```

MOVE@

Purpose To copy an array of N bytes.

Syntax
```
SUBROUTINE MOVE@(FROM,TO,N)
```

Description This routine copies N bytes of data from FROM to TO. No data conversion is performed and *no* checks are made to ensure that the source and destination are large enough (even in /CHECK mode). Arguments FROM and TO may be of any type, N must be INTEGER (KIND=1), (KIND=2) or (KIND=3).

NEWLINE@

Purpose To write a carriage return/linefeed to the screen (standard output).

Syntax
```
SUBROUTINE NEWLINE@
```

NONBLK

Purpose To obtain the position of the first non-blank character.

Syntax
```
INTEGER (KIND=2) FUNCTION NONBLK(A)
CHARACTER (LEN=*) A
```

Return value NONBLK returns the position of the first non-blank character in the character argument A. If the argument is wholly blank, 0 is returned.

Notes For new code, the standard Fortran 95 intrinsic function VERIFY should be used instead.

OPENCOMMDEVICE@

Purpose To open a serial port for I/O.

Syntax
```
INTEGER (KIND=3)
OPENCOMMDEVICE@(PORTNUM,COMSPEC,RSIZE,TSIZE)
INTEGER (KIND=3) PORTNUM, RSIZE,TSIZE
CHARACTER (LEN=*) COMSPEC
```

Description To initiate serial communications between the computer and external devices a communications port must be selected and opened. On a standard PC there is a maximum of four serial ports, although it is common for only two to be installed. PORTNUM can therefore be either 1, 2, 3 or 4. Port 1 is commonly used to connect the mouse and so may not be available.
COMSPEC is a string that specifies the baud rate, parity, data and stop bit information (e.g. '9600, n, 8, 1'). Possible values are:

baud rate:	300,600,1200,2400,4800,9600,19200,38400,57600,115200
parity:	n (none), o (odd), e (even)
data bits:	7 or 8
stop bits:	0 or 1

Under Windows 3.1(1) it is necessary to specify a size for input and output buffers. The size of the input buffer RSIZE and the size of the output buffer TSIZE should be set at about 1024. On slower systems with high data rates it may be advisable to specify larger values.

Return value OPENCOMMDEVICE@ returns a positive value when successful otherwise it returns -1.

OPENF@

Purpose To open a file for reading or writing with sharing attributes.

Syntax

```
SUBROUTINE OPENF@(FILE,ATTR,HANDLE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) ATTR,HANDLE,ERROR_CODE
```

Description This routine opens the given file FILE for reading or writing without deleting existing data. If the file does not exist then it is created. The subroutine returns the file handle HANDLE for use with other file handling routines in this chapter. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

ATTR is an input to the subroutine that is used to specify the attributes of the open file. The first four bits (the least significant or rightmost) specify the access mode. The next four bits specify the share mode as follows.

Access mode	read only	Z'00'	Current application can only read the file.
	write only	Z'01'	Current application can only write to the file.
	read/write	Z'02'	
Share mode	compatibility mode	Z'00'	Compatible with systems that do not allow sharing, equivalent to ‘deny all’.
	deny all	Z'10'	Other applications can not read or write to the file whilst it is open in the current application.
	deny write	Z'20'	Other applications can not write to the file whilst it is open in the current application.
	deny read	Z'30'	Other applications can not read the file whilst it is open in the current application.
	deny none	Z'40'	Other applications are allowed to read or write to the file.

Thus for read/write access with shared ‘deny all’ access use Z'12'.

Notes The sharing attributes of an open file are determined by the first application to open it. So if it is already open the current sharing mode will have no effect.

HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see chapter 8 for further details).

OPENR@

Purpose To open a file for reading.

Syntax
```
SUBROUTINE OPENR@(FILE,HANDLE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description This routine opens the given file FILE for reading and returns the file handle HANDLE for use with other file handling routines. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see chapter 8 for further details).

Example See OPENRW@

OPENRW@

Purpose To open a file for reading or writing.

Syntax
```
SUBROUTINE OPENRW@(FILE,HANDLE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description This routine opens a file FILE for reading or writing and returns the file handle HANDLE for use with other file handling routines. If the file does not exist it is created, however an existing file is *not* emptied and may be over-written at the current position. If the intended action depends on whether or not a given file exists, then a prior call to OPENR@ can be used to test if it does exist. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see chapter 8 for further details).

Example
```
!    Run the program and list TEST.DAT after each run.
     PROGRAM READFILE
      INTEGER (KIND=2)::HANDLE,ERROR_CODE
      INTEGER (KIND=3)::BYTES
      CHARACTER (LEN=80)::LINE
```

```
  CALL OPENR@('TEST.DAT',HANDLE,ERROR_CODE)
  IF(ERROR_CODE /= 0) THEN
    CALL OPENW@('TEST.DAT',HANDLE,ERROR_CODE)
    CALL WRITEFA@('Test data.....',HANDLE,ERROR_CODE)
  ELSE
    CALL CLOSEF@(HANDLE,ERROR_CODE)
    CALL OPENRW@('TEST.DAT',HANDLE,ERROR_CODE)
    LINE=' '
    CALL READFA@(LINE,HANDLE,BYTES,ERROR_CODE)
    CALL DOSERR@(ERROR_CODE)
    WRITE(*,*) LINE
    CALL WRITEFA@('More info.....',HANDLE,ERROR_CODE)
  ENDIF
END PROGRAM READFILE
```

OPENW@

Purpose To open a file for writing.

Syntax

```
SUBROUTINE OPENW@(FILE,HANDLE,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description This routine opens a file FILE for writing, by creating a file or emptying the file if it already exists. It returns the file handle HANDLE for use with other file handling routines. ERROR_CODE is returned as zero if the operation has succeeded, otherwise it is returned with the relevant system error code.

Notes HANDLE can also be obtained by using the standard Fortran routine OPEN followed by INQUIRE together with FUNIT=<filehandle> (see chapter 8 for further details).

Example See OPENRW@

PERMIT_UNDERFLOW@

Purpose To switch on floating point underflow checking.

Syntax

```
SUBROUTINE PERMIT_UNDERFLOW@(PERMISSION)
LOGICAL (KIND=2) PERMISSION
```

Description If PERMISSION is .FALSE. then subsequent underflows will force a program fault. If PERMISSION is .TRUE. then subsequent floating point underflows return zero.

By default underflow is permitted.

Note A subsequent call to MASK_UNDERFLOW@ has the effect that underflows will not generate an exception with the result that the run time library cannot generate an error condition if an underflow occurs. See chapter 22.

POP@

Purpose To pop a value off the system stack.

Syntax
```
SUBROUTINE POP@(A)
INTEGER (KIND=3) A
```

Description This routine is the opposite of PUSH@.

PRERR@

Purpose To print the error message associated with a given error code.

Syntax
```
SUBROUTINE PRERR@(ERROR_CODE,STRING)
INTEGER (KIND=2) ERROR_CODE
CHARACTER (LEN=*) STRING
```

Description This routine does nothing if ERROR_CODE is zero, otherwise it prints the user-supplied string STRING followed by the system error message corresponding to ERROR_CODE. The routine returns normally and program execution continues. ERROR_CODE will normally be a value returned by an earlier call to a routine that could generate a system error condition.

PRINT_BYTES@

Purpose To write a sequence of hexadecimal values.

Syntax
```
SUBROUTINE PRINT_BYTES@(IARR,N)
INTEGER (KIND=2) IARR(*),N
```

Description PRINT_BYTES@ writes N bytes of datum IARR in hexadecimal to standard output, separating each byte value by a space, with no terminating new line.

PRINT_BYTES_R@

Purpose To write a hexadecimal sequence in reverse order.

Syntax
```
SUBROUTINE PRINT_BYTES_R@(IARR,N)
INTEGER (KIND=2) IARR(*),N
```

Description This routine is similar to PRINT_BYTES@, but the bytes are written in reverse order.

PRINT_HEX1@

Purpose To print a 1 byte hexadecimal number (2 digits) without a new line.

Syntax
```
SUBROUTINE PRINT_HEX1@(L)
INTEGER (KIND=1) L
```

PRINT_HEX2@

Purpose To print a 2 byte hexadecimal number (4 digits) without a new line.

Syntax
```
SUBROUTINE PRINT_HEX2@(L)
INTEGER (KIND=2) L
```

PRINT_HEX4@

Purpose To print a 4 byte hexadecimal number (8 digits) without a new line.

Syntax
```
SUBROUTINE PRINT_HEX4@(L)
INTEGER (KIND=3) L
```

PRINT_I1@

Purpose To print an INTEGER (KIND=1) decimal number without a new line.

Syntax
```
SUBROUTINE PRINT_I1@(I)
INTEGER (KIND=1) I
```

PRINT_I2@

Purpose To print an INTEGER (KIND=2) decimal number without a new line.

Syntax
```
SUBROUTINE PRINT_I2@(I)
INTEGER (KIND=2) I
```

PRINT_I4@

Purpose To print an INTEGER (KIND=3) decimal number without a new line.

Syntax
```
SUBROUTINE PRINT_I4@(L)
INTEGER (KIND=3) L
```

PRINT_R4@

Purpose To print a REAL (KIND=1) value as a number without a new line.

Syntax
```
SUBROUTINE PRINT_R4@(R)
REAL (KIND=1) R
```

PRINT_R8@

Purpose To print a REAL (KIND=2) value as a number without a new line.

Syntax
```
SUBROUTINE PRINT_R8@(R)
REAL (KIND=2) R
```

PUSH@

Purpose To push a value on the system stack.

Syntax
```
SUBROUTINE PUSH@(A)
INTEGER (KIND=3) A
```

Description The argument A is pushed on to the system stack. Two bytes will be pushed for INTEGER (KIND=2), four for INTEGER (KIND=3) etc. Values saved in this way can be restored again with POP@. The corresponding POP@ call must have an argument of the same type (otherwise the wrong number of bytes would be popped). A routine may return with data still pushed on the stack - such data is then lost.

QUIT_CLEANUP@

Purpose To print a message and exit from a program with Control-break

Syntax

```
SUBROUTINE QUIT_CLEANUP@(MESSAGE)
CHARACTER (LEN=*) MESSAGE
```

Description This routine uses SET_TRAP@ to trap Control-break. The system responds to Control-break by printing the given message and returning to DOS. This routine provides a simple way to enable programs to be terminated early in a graceful fashion.

Example

```
CALL QUIT_CLEANUP@('Quit pressed - program abandoned')
```

RANDOM

Purpose To return a pseudo-random double precision value.

Syntax

```
REAL (KIND=2) FUNCTION RANDOM()
```

Description This routine sets its seed automatically and produces the same sequence every time the program is run.

Alternatively, you may use DATE_TIME_SEED@ or SET_SEED@ which are described below.

Return value RANDOM returns a uniformly distributed random number x such that $0.0D0 < x \leq 1.0D0$.

Example

```
REAL (KIND=2)::RANDOM,RANVEC(100)
DO I=1,100
  RANVEC(I)=RANDOM()
END DO
. . .
```

READCOMMDEVICE@

Purpose To read data from an open serial port.

Syntax
```
INTEGER (KIND=3) READCOMMDEVICE@(PORTNUM,STRING,NREAD)
INTEGER (KIND=3) PORTNUM,NREAD
CHARACTER (LEN=*) STRING
```

Description Use this function to read data from a serial port that has been opened by a call to OPENCOMMDEVICE@. PORTNUM must be a valid port number in the range 1 to 4 and NREAD is set as the maximum number of characters to be read. After the call STRING will contain the data held in the serial port up to NREAD characters. Fewer than NREAD characters will be read if one of the termination characters (set using SETCOMMTERMINATECHAR@) is encountered.

Return value READCOMMDEVICE@ returns the number of characters read or -1 if an error occurred.

READF@

Purpose To read binary data from a file.

Syntax
```
 SUBROUTINE READF@(DATA,HANDLE,NBYTES,NBYTES_READ, &
& ERROR_CODE)
 CHARACTER (LEN=*) DATA
 INTEGER (KIND=2) HANDLE,ERROR_CODE
 INTEGER (KIND=3) NBYTES,NBYTES_READ
```

Description This routine reads NBYTES of data from an open file with a given HANDLE. ERROR_CODE is returned as zero for success or a system error code. If end of file is reached, NBYTES_READ is returned as -1, with an ERROR_CODE of zero. Also NBYTES_READ may be returned as less than NBYTES. This routine should be used on binary data.

Notes If the input value of NBYTES is supplied as a constant and the compiler option /INTS is used, it will be necessary to force the length of the constant to 4 bytes.

READFA@

Purpose To read ASCII text from a file.

Syntax
```
SUBROUTINE READFA@(DATA,HANDLE,NBYTES_READ,ERROR_CODE)
CHARACTER (LEN=*) DATA
INTEGER (KIND=2) HANDLE,ERROR_CODE
INTEGER (KIND=3) NBYTES_READ
```

Description READFA@ reads a line of text from an open file with a given HANDLE. Tabs are expanded as necessary. ERROR_CODE is returned as zero for success or with a system error code. If end of file is reached, NBYTES_READ is returned as -1, with an ERROR_CODE of zero.

Example See OPENRW@

RENAME@

Purpose To rename a file.

Syntax
```
SUBROUTINE RENAME@(FILE1,FILE2,ERROR_CODE)
CHARACTER (LEN=*) FILE1,FILE2
INTEGER (KIND=2) ERROR_CODE
```

Description RENAME@ renames FILE1 as FILE2 in exactly the same way as the DOS RENAME command.

RESTORE_DEFAULT_HANDLER@

Purpose Remove a user defined exception handler.

Syntax
```
SUBROUTINE RESTORE_DEFAULT_HANDLER@(EXCEPTION)
INTEGER (KIND=3) EXCEPTION
```

Description Remove the default exception handler for the exception event given by EXCEPTION and re-install the default handler.

RETURN_STORAGE@

Purpose To return a block of storage.

Syntax
```
SUBROUTINE RETURN_STORAGE@(ADDR)
INTEGER (KIND=3) ADDR
```

Description Returns a block of storage previously allocated by one of the storage management routines. ADDR must be the address of the start of the storage block to be returned.

RFPOS@

Purpose To get the position of a file.

Syntax
```
SUBROUTINE RFPOS@(HANDLE,POSITION,ERROR_CODE)
INTEGER (KIND=2) HANDLE,ERROR_CODE
INTEGER (KIND=3) POSITION
```

Description This routine returns the POSITION of the file open on the given HANDLE. ERROR_CODE is returned as zero or with a system error code if it fails.

RSORT@

Purpose To sort a REAL (KIND=1) array.

Syntax
```
SUBROUTINE RSORT@(A,R,N)
REAL (KIND=1) R(N)
INTEGER (KIND=3) A(N),N
```

Description RSORT@ sorts the REAL array R by setting pointers from 1 to N in the array A in the same manner as CHSORT@.

RUNERR@

Purpose To print the run-time error corresponding to a given IOSTAT value.

Syntax
```
SUBROUTINE RUNERR@(ISTAT)
INTEGER (KIND=2) ISTAT
```

Description RUNERR@ prints an error message on the screen corresponding to a given IOSTAT value ISTAT. These are the error messages listed in chapter 21.

SAYINT

Purpose To return an integer argument as text.

Syntax
```
CHARACTER (LEN=*) FUNCTION SAYINT(I)
INTEGER (KIND=3) I
```

Description As an example, the value I=-270 would return the character value 'MINUS TWO HUNDRED AND SEVENTY'.

Example
```
PROGRAM SAY
 CHARACTER (LEN=80)::SAYINT
 INTEGER (KIND=3)::I
  DO I=10,0,-1
    PRINT *,SAYINT(I)
  END DO
PRINT *,'we have lift off!'
END PROGRAM SAY
```

SECONDS_SINCE_1970@

Purpose To get the number of seconds from a fixed date.

Syntax
```
SUBROUTINE SECONDS_SINCE_1970@(DR)
REAL (KIND=2) DR
```

Description Returns the value of DR as the number of seconds that have elapsed since 12.00am on 1st January 1970.

SECONDS_SINCE_1980@

Purpose To get the number of seconds from a fixed date.

Syntax
```
SUBROUTINE SECONDS_SINCE_1980@(DR)
REAL (KIND=2) DR
```

Description This routine returns the value of DR as the number of seconds that have elapsed since

12.00am on 1st January 1980. It can be used in a similar way to CLOCK@ (or DCLOCK@). SECONDS_SINCE_1980@ should be used when making timings which straddle midnight.

SET_BIT@

Purpose To set the N'th bit of an array.

Syntax

```
SUBROUTINE SET_BIT@(IA,N)
INTEGER (KIND=2) IA(*),N
```

Description Sets the N'th bit of the array IA. N can be INTEGER (KIND=1), (KIND=2) or (KIND=4) and IA can be of any datatype.

SET_FILE_ATTRIBUTE@

Purpose To set a file attribute.

Syntax

```
SUBROUTINE SET_FILE_ATTRIBUTE@(FILE,IAT,ERROR_CODE)
CHARACTER (LEN=*) FILE
INTEGER (KIND=2) IAT,ERROR_CODE
```

Description This routine sets the attributes of the file FILE to IAT. ERROR_CODE is returned as zero for success or it is the system error code. This routine is useful for performing such tasks as changing the read-only status of a file, hiding or revealing a file, reseting the backup bit etc..

The following program will read a file name (possibly including wild cards) from the terminal and make the files read-only.

Example

```
CHARACTER (LEN=20)::FILE,FILES(1000),CMNAM
INTEGER (KIND=2)::ATTR(1000),DATE(1000),TIME(1000)
INTEGER (KIND=3)::FILE_SIZE(1000)
CALL FILES@(CMNAM(),N,1000,FILES,ATTR,DATE,TIME,FILE_SIZE)
DO I=1,N
  CALL SET_FILE_ATTRIBUTE@(FILES(I),OR(ATTR(I),1),IC)
  CALL DOSERR@(IC)
END DO
```

SET_SEED@

Purpose To enter a new "seed" for the pseudo-random number generator function RANDOM.

Syntax
```
SUBROUTINE SET_SEED@(SEED)
REAL (KIND=2) SEED
```

Description This routine sets the seed for the random number generator to a value based on SEED. SEED may take any value. Each value produces a repeatable sequence of pseudo-random numbers.

SET_SUFFIX@

Purpose To change the extension of a given file name.

Syntax
```
SUBROUTINE SET_SUFFIX@(FILENAME,SUFFIX,L)
CHARACTER (LEN=*) FILENAME
CHARACTER (LEN=3) SUFFIX
LOGICAL (KIND=2) L
```

Description SET_SUFFIX@ changes the file extension of a given file with name FILENAME. SUFFIX is the new extension required, given without the period ("."). The value L will be set to .TRUE. if the file had an extension that was not SUFFIX. L will be given the value .FALSE. if the file had the same or no extension.

Example
```
      A='c:\ftn95.dir\file.dat'
      CALL SET_SUFFIX@(A,'ASC',L)
!  At this point A contains 'c:\ftn95.dir\file.ASC'
!  and L contains .TRUE.
```

SET_SUFFIX1@

Purpose To add an extension to a given file name.

Syntax
```
SUBROUTINE SET_SUFFIX1@(FILENAME,SUFFIX,L)
CHARACTER (LEN=*) FILENAME
CHARACTER (LEN=3) SUFFIX
LOGICAL (KIND=2) L
```

Description SET_SUFFIX1@ will add a file-extension SUFFIX to the string FILENAME containing a filename if none is present. The filename will be left as it is if the

filename already contains an extension. The extension should be given without the period (“.”). The value L will be set to .TRUE. if the file had an extension that was not SUFFIX. L will be given the value .FALSE. if the filename had the same or no extension.

Example

```
      A='c:\ftn95.dir\file.dat'
      CALL SET_SUFFIX1@(A,'ASC',L)
!  At this point A contains 'c:\ftn95.dir\file.dat'
!  and L contains .TRUE.
```

SETCOMMTERMINATECHAR@

Purpose To set the characters that may be used to terminate a call to READCOMMDEVICE@ .

Syntax

```
INTEGER (KIND=3) SETCOMMTERMINATECHAR@(PORTNUM,STRING,LEN)
INTEGER (KIND=3) PORTNUM,LEN
CHARACTER (LEN=*) STRING
```

Description PORTNUM must be a valid serial port number in the range 1 to 4. STRING contains a string of length LEN. This string is used to provide a list of characters that may be used to terminate a call to READCOMMDEVICE@. The termination character is discarded when READCOMMDEVICE@ is called. For example `char(0)//char(12)//char(15)` provides for the data to be terminated by a null character, a line feed or a carriage return.

Return value SETCOMMTERMINATECHAR@ returns a positive value if successful or -1 if an error has occurred.

SETECHOONREADCOMM@

Purpose To set the communication port to echo back to the sending device.

Syntax

```
INTEGER (KIND=3) SETECHOONREADCOMM@(PORTNUM,STATE)
INTEGER (KIND=3) PORTNUM,STATE
```

Description When communicating with a serial device such as a terminal, it is often necessary to return the data to the sender. In the case of a terminal the data will be transmitted from its keyboard to the host computer. SETECHOONREADCOMM@ is used to enable (STATE = 1) or disable (STATE = 0) the echoing of data. Data that is echoed back to a terminal will be displayed on its VDU. PORTNUM is a valid port number in the range 1 to 4.

Return value SETECHOONREADCOMM@ returns a positive value if successful or -1 if an error has occurred.

SHRINK_STORAGE@

Purpose To shrink a block of storage.

Syntax
```
SUBROUTINE SHRINK_STORAGE@(ADDR,N)
INTEGER (KIND=3) ADDR,N
```

Description Shrinks a block of storage previously allocated by one of the storage management routines. ADDR must be the address of the start of the storage block whose size is to be adjusted. N is the new size of the block. This routine cannot be used to enlarge a storage block.

SLEEP@

Purpose To suspend program execution for a specified time interval.

Syntax
```
SUBROUTINE SLEEP@(TIME)
REAL (KIND=1) TIME
```

Description The time is given in seconds and is accurate to within one tick (18.2 ticks per second).

SOU@

Purpose To output text with a new line, omitting any trailing blanks.

Syntax
```
SUBROUTINE SOU@(A)
CHARACTER (LEN=*) A
```

Description SOU@ writes to the screen from the character argument A. It is identical to COU@ except that the cursor will not scan over any trailing blanks.

See also COU@, COUA@, SOUA@.

SOUA@

Purpose To output text without a new line, omitting any trailing blanks.

Syntax
```
SUBROUTINE SOUA@(A)
CHARACTER (LEN=*) A
```

Description SOUA@ writes to the screen from the character argument A omitting any trailing blanks. It is like SOU@ but does not output a new line.

See also COU@, COUA@, SOU@.

START_PROCESS@

Purpose To create a new process and wait for it to terminate.

Syntax
```
INTEGER FUNCTION START_PROCESS@(COMMAND,PARAMS)
CHARACTER (LEN=*) COMMAND,PARAMS
```

Description COMMAND is the command line that is to be executed. PARAMS is a string of parameters that is appended to the command line.

Return value Returns 0 or a positive value if successful. A return value of -1 indicates an error condition.

Notes The new process is given the same 'show' state as that used when creating the calling process. This will usually be 'show normal' or 'show maximised'.

An alternative function called START_PPROCESS@ takes the same form but returns immediately (i.e. it does not wait for the created process to terminate).

Example
```
INTEGER START_PROCESS@,i
i=START_PROCESS@('run.exe',' ')
END
```

TEMP_FILE@

Purpose To provide a unique name for a file.

Syntax
```
SUBROUTINE TEMP_FILE@(FILEX,ERROR_CODE)
CHARACTER (LEN=*) FILEX
```

```
INTEGER (KIND=2) ERROR_CODE
```

Description TEMP_FILE@ provides a name which may be used for the creation of a temporary file. This name (of the form F$dddddd.TMP where d is a digit) is different from all the file names within the current directory. It is important to note that this routine does not create or open a file.

TEMP_PATH@

Purpose To get a suitable name for a temporary file.

Syntax

```
SUBROUTINE TEMP_PATH@(PATH)
CHARACTER (LEN=*) PATH
INTEGER (KIND=2) ERROR_CODE
```

Description This routine is obsolete. Use TEMP_FILE@ instead.

TEMP_PATH@ makes up a path name for a temporary file. The file name component is created in the same way as for TEMP_FILE@. The directory is that given by the TMPDIR environment variable (or "\TMP" if this is not set). Note that this routine does not actually open the file.

TEST_BIT@

Purpose To test if the N'th bit of an array is set.

Syntax

```
INTEGER (KIND=2) FUNCTION TEST_BIT@(IA,N)
INTEGER (KIND=2) IA(*),N
```

Description TEST_BIT@ may be declared as LOGICAL or INTEGER.

Return value TEST_BIT@ returns 1 or 0 (.TRUE. or .FALSE.) according to whether the N'th bit of IA is set or not.

TIME@

Purpose To get the time in the format HH:MM:SS.

Syntax

```
CHARACTER*8 FUNCTION TIME@()
```

TODATE@

Purpose To convert the time given to a date for the form MM/DD/YY (American format).

Syntax
```
CHARACTER (LEN=*) FUNCTION TODATE@(SECS)
INTEGER (KIND=3) SECS
```

TOEDATE@

Purpose To convert the time given to a date in the format DD/MM/YY (European format).

Syntax
```
CHARACTER (LEN=*) FUNCTION TOEDATE@(SECS)
INTEGER (KIND=3) SECS
```

TOFDATE@

Purpose To get the date in text form.

Syntax
```
CHARACTER (LEN=*) FUNCTION TOFDATE@(SECS)
```

Return value TOFDATE@ returns the date in textual format based on time given in the form: “Friday January 29, 1993”.

TOTIME@

Purpose To return the time in the form “HH:MM:SS”.

Syntax
```
CHARACTER (LEN=*) FUNCTION TOTIME@(SECS)
INTEGER (KIND=3) SECS
```

Return value Returns the time corresponding to SECS.

TRANSLATE_FORTRAN_HANDLE@

Purpose To change from a Salford file handle to a Win32 file handle.

Syntax
```
INTEGER (KIND=3) TRANSLATE_FORTRAN_HANDLE@(HANDLE)
INTEGER (KIND=2) HANDLE
```

Return value Returns the Win32 file handle corresponding to HANDLE. HANDLE is a handle returned by OPENR@, OPENRW@, or OPENF@.

TRAP_EXCEPTION@

Purpose Install a user defined exception handler

Syntax
```
INTEGER(3) FUNCTION TRAP_EXCEPTION@(EXCEPTION, ROUTINE)
INTEGER(3) EXCEPTION, ROUTINE
EXTERNAL ROUTINE
```

Description The function ROUTINE is installed as the default method of handling the event specified by EXCEPTION.

Return value If EXCEPTION is a valid exception event, the location of the previous handler is returned. 0 is returned if EXCEPTION is an invalid exception code.

See chapter 22 for further details.

TRIM@

Purpose To remove leading blanks.

Syntax
```
SUBROUTINE TRIM@(X)
CHARACTER (LEN=*) X
```

Description TRIM@ is used to remove leading blank characters from the character argument X.

Notes For new code, the standard Fortran 95 intrinsic function TRIM should be used instead.

TRIMR@

Purpose To rotate a character string right until there are no trailing blanks.

Syntax
```
SUBROUTINE TRIMR@(X)
CHARACTER (LEN=*) X
```

Notes If the string is blank, it is left unchanged.

For new code, the standard Fortran 95 intrinsic function ADJUSTR should be used

instead.

UNDERFLOW_COUNT@

Purpose To get the number of floating point underflows.

Syntax
```
SUBROUTINE UNDERFLOW_COUNT@(COUNT)
INTEGER (KIND=3) COUNT
```

Description COUNT is returned as the number of underflows that have occurred since the start of the program.

UPCASE@

Purpose To alter a character argument so that all letters become upper case.

Syntax
```
SUBROUTINE UPCASE@(A)
CHARACTER (LEN=*) A
```

Example
```
PROGRAM UCASE
   CHARACTER (LEN=10)::FRED
  FRED = 'ABcd'
  CALL UPCASE@(FRED)
  IF(FRED /= 'ABCD') PRINT *,'UPCASE@ routine has failed'
END PROGRAM UCASE
```

WILDCHECK@

Purpose To check for the matching of a file name with a wild card.

Syntax
```
LOGICAL (KIND=2) FUNCTION WILDCHECK@(WILDCARD,NAME)
CHARACTER (LEN=*) WILDCARD,NAME
```

Description Returns .TRUE. if NAME can be matched with WILDCARD, .FALSE. otherwise (including when the syntax of WILDCARD or NAME is invalid).

WRITECOMMDEVICE@

Purpose To write a string to a serial port.

Syntax
```
INTEGER (KIND=3) WRITECOMMDEVICE@(PORTNUM,STRING)
INTEGER (KIND=3) PORTNUM
CHARACTER (LEN=*) STRING
```

Description Use this function to write a string to a serial port that has been opened by a call to OPENCOMMDEVICE@. PORTNUM must be a valid port number in the range 1 to 4.

Return value The function returns the number of characters written or -1 if an error has occurred.

WRITEF@

Purpose To write binary data to a file.

Syntax
```
SUBROUTINE WRITEF@(DATA,HANDLE,NBYTES,ERROR_CODE)
CHARACTER (LEN=*) DATA
INTEGER (KIND=2) HANDLE,ERROR_CODE
INTEGER (KIND=3) NBYTES
```

Description Writes NBYTES of binary data DATA to the file with the given handle. ERROR_CODE is returned as zero for success or a system error code on failure. No data compression on insertion of control characters is performed.

Notes If the input value of NBYTES is supplied as a constant, it is usually necessary to force its length to 4 bytes (append L to the decimal value).

WRITEFA@

Purpose To write a line of data to an ASCII file.

Syntax
```
SUBROUTINE WRITEFA@(DATA,HANDLE,ERROR_CODE)
CHARACTER (LEN=*) DATA
INTEGER (KIND=2) HANDLE,ERROR_CODE
```

Description WRITEFA@ writes DATA to an open file with a given HANDLE. A carriage return/linefeed is added to the end of the data. ERROR_CODE is returned as zero for success, otherwise it returns the system error code.

Example

```
INTEGER (KIND=2)::HANDLE,ERROR_CODE
CALL OPENW@('TEST.DAT',HANDLE,ERROR_CODE)
CALL DOSERR@(ERROR_CODE)
CALL WRITEFA@('Test data.....',HANDLE,ERROR_CODE)
CALL WRITEFA@('More info.....',HANDLE,ERROR_CODE)
```

26.

Salford environment variables

The following environment variables can be used with Salford compilers in order to over-ride the default settings. Depending on the operating system, environment variables are defined in an AUTOEXEC.BAT file or by using the Control Panel.

MOD_PATH

MOD_PATH can be used in association with FTN95 in order to specify an additional search path for .MOD and associated .OBJ module files. For example,

SET MOD_PATH=C:\FTN95;C:\DBOS.DIR\INCLUDE

would result in FTN95 searching in these directories if the .MOD file could not be found in the current working directory.

F95INCLUDE

F95INCLUDE is like MOD_PATH and is used with FTN95 in order to specify an alternative search path for INCLUDE files that are presented between angled brackets. For example,

INCLUDE <WINDOWS.INS>

SALFORD_COLOUR

SALFORD_COLOUR is used switch on and/or change the default colours that are associated with the compiler command line option /COLOUR.

To switch on colouring use:

SET SALFORD_COLOUR=*

To switch on and change the default colours use the form:

SET SALFORD_COLOUR=<e>[;<w>[;<c>[;<s>[;<l>]]]]

The symbols represent:

- <e> Error messages
- <w> Warning messages
- <c> Comment messages
- <s> Summary lines
- <l> Program source in error reports

Each of these takes one of the following:

k	blacK
b	Blue
g	Green
c	Cyan
r	Red
m	Magenta
y	Yellow
w	White
*	Use default value

A preceding hyphen indicates that high intensity will be used.

For example:

```
SET SALFORD_COLOUR=-r;r;y;-w;-y
```

If this environment variable is not set then the compiler will use "-r;-y;-g;w;-w" with /COLOUR.

DATE_FAIL

If DATEFAIL=YES then a runtime error will occur if either of the Salford library functions DATE@ and EDATE@ is used to return a date that is not year 2000 compliant (i.e. the 8 character form is used rather than the 10 character form).

TMPDIR etc.

TMPDIR can be used to set the directory to be used with OPEN(STATUS='SCRATCH') and the Salford library routines TEMP_FILE@ and TEMP_PATH@. Other environment variables are also available as indicated in the following search order:

%TMPDIR%
%TMP%
%TEMP%
%HOMEDRIVE%+%HOMEPATH%
\TMP
\TEMP
current working directory.

SALFENVAR

SALFENVAR can be set to MASK_UNDERFLOW or UNMASK_UNDERFLOW in order to provide a default state for masking underflows (see page 232 for details).

LIB

LIB is used by SLINK and provides a search path for library files.

SCCINCLUDE

SCCINCLUDE is used to provide a search path for C/C++ include files to be used with the Salford C/C++ compiler.

SOURCEPATH

SOURCEPATH is used by the Salford debuggers to provide an alternative search path for source files. See page 33 for further details.

LINK_SUBSYSTEM

This variable is used by SLINK. Setting

LINK_SUBSYSTEM=3

causes SLINK to use the Windows 3.10 subsystem. This affects the Windows style of a graphics user interface. The default is Windows 4.0.

Index

D

E

F

N

O

P

Q

R

S