SUN/143.4

Starlink Project
Starlink User Note 143.4

P M Allan
A J Chipperfield

20 July 2001

FIO/RIO
FORTRAN file I/O routines
Version 1.5
Programmer's Manual

Abstract

FIO/RIO is a subroutine package that allows a FORTRAN programmer to access sequential and direct access data files in a machine independent manner. The package consists of stand alone FIO and RIO routines, which can be used independently of the Starlink software environment, plus routines to interface to the Starlink parameter system.

Contents

1 Introduction

The FIO/RIO package is intended for handling record oriented files (e.g. simple text files) in both ADAM tasks and stand-alone FORTRAN programs. Although most bulk data will be stored in HDS files, there are occasions when the use of HDS is not appropriate. Writing formatted sequential files that are intended for printing as reports is one obvious example. When it is necessary to read and write record oriented files, then the use of FIO can ease the writing of such programs and will assist in the production of portable software.

The essential difference between the FIO and RIO routines is that FIO handles sequential files and RIO handles direct access files (also known as random access files, hence the R in RIO). The FIO routines are primarily intended for handling formatted, sequential files, but some can also process unformatted, sequential files. Formatted, sequential access files may have the first character of each record interpreted as a carriage control character when the file is printed. Whether or not a formatted file contains carriage control characters can be specified when the file is created. RIO routines are primarily used to handle unformatted, direct access files, although some can handle formatted, direct access files as well.

FIO and RIO use a common table of file descriptors so that file descriptors created by RIO routines may be used with appropriate FIO routines: e.g. FIO_FNAME returns the filename associated with a file descriptor obtained via either FIO or RIO.

The normal Starlink 'inherited status' error handling strategy is employed throughout. Any FIO/RIO routine that fails will report an error and set the STATUS argument to an appropriate value. Symbolic constants for these STATUS values are given in appendix E.

2 FIO/RIO descriptors

FIO/RIO uses internal file descriptors to maintain information about the files that it processes. The descriptors contain the FORTRAN unit number of the file, the name of the file, the access mode and the record size. Knowledge of the access mode allows FIO/RIO to check for invalid operations, such as writing to a read-only file. Checking for invalid I/O operations before they are actually performed makes programs more robust, since the corresponding I/O error is never generated.

The FIO/RIO file descriptors do not contain any more information about a file than could be obtained by using the FORTRAN INQUIRE statement, but they store the information in such a way that it is more efficient to use descriptors than the INQUIRE statement.

3 Using FIO/RIO

FIO/RIO can be used in three main ways; you can use it in a minimalist way to ease the writing of normal FORTRAN programs, you can use the extra functionality provided by FIO file descriptors in stand alone FORTRAN programs, or you can use the ADAM parameter system interface in ADAM programs.

3.1 Routines to enhance simple FORTRAN I/O

Some of the FIO/RIO routines do not use FIO file descriptors and are provided to simplify common I/O operations. For example, FIO_GUNIT will get an unused FORTRAN unit number. Using this routine is better than 'hard wiring' unit numbers into code as you may not know what unit numbers other subroutines are using. The routines that do not use the FIO file descriptors are:

FIO_ERASE Erase a file

FIO_GUNIT Get a FORTRAN I/O unit number

FIO_PUNIT Return an FORTRAN I/O unit number

FIO_REP Report an I/O error

FIO_SERR Report an I/O error

FIO_TEST Test if a status value belongs to a certain class of errors

RIO_ERASE Erase a file

Here is an example of the use of some of these routines.

```
      ...
*  Get a unit number.
      CALL FIO_GUNIT( UNIT, STATUS )
*  Open a file.
      OPEN( UNIT=UNIT, FILE=FILNAM, STATUS='NEW', IOSTAT=ISTAT )
      IF ( IOSTAT .EQ. 0 ) THEN
*  Save the data.
         WRITE( UNIT, '(5F10.2)' ) ( X( I ), I = 1, 5 )
         CLOSE( UNIT )
      ELSE
*  Report an error
         CALL FIO_REP( UNIT, FILNAM, ISTAT, ' ', STATUS )
      END IF
*  Return the unit number.
      CALL FIO_PUNIT( UNIT, STATUS )
      ...
```

Consistent use of the FIO_GUNIT and FIO_PUNIT routines has reduced the likelihood of a clash of unit number between this part of the program and some other part, and the use of FIO_REP allows machine independent reporting of any errors.

3.2 The stand-alone subroutines

In addition to the routines in the previous section, FIO provides a set of routines to do some simple I/O on files. FIO maintains a set of file descriptors for active files which are used by these routines. These descriptors contain such things as the access mode of a file (read only, update, etc.), which allow FIO to trap some errors rather than permitting a run time error to

occur. For example, if an attempt is made to write to a file that has been opened with 'read only' access, FIO will report the error, but the program will not crash, allowing the user to take corrective action. Use of these routines also makes user written code more portable. Issues such as requiring CARRIAGECONTROL='LIST' in DEC FORTRAN OPEN statements are handled internally. The routines that handle FIO file descriptors are:

FIO_CLOSE Close a file.

FIO_FNAME Get the name of a file.

FIO_OPEN Open a file.

FIO_READ Read a file.

FIO_READF Read a file (faster than FIO_READ).

FIO_RWIND Rewind a file.

FIO_UNIT Get the unit number of a file.

FIO_WRITE Write a file.

RIO_CLOSE Close a file.

RIO_OPEN Open a file.

RIO_READ Read a file.

RIO_WRITE Write a file.

Note that the same file descriptors are used by the FIO and RIO routines, so these can be freely mixed, where appropriate.

Here is an example of the use of some of these routines.

```
      ...
*  Open a file.
      CALL FIO_OPEN( FILNAM, 'WRITE', 'LIST', 0, FD, STATUS )
*  Write the data.
         DO I = 1, N
            CALL FIO_WRITE( FD, BUF( I ), STATUS )
         END DO
*  Close the file.
      CALL FIO_CLOSE( FD, STATUS )
      ...
```

Note that there is no testing for errors in this piece of code since the FIO routines follow the normal Starlink convention for error handling and will not execute if STATUS is bad. However, if the loop is to be executed many times, it would be worth testing that the call to FIO_OPEN was successful, otherwise you could end up executing the loop many times to no effect.

3.3 The environment level routines

The last way of using FIO/RIO is in its fully integrated ADAM form. The following routines provide an interface to the ADAM parameter system:

FIO_ANNUL Annul a file descriptor and close the file.

FIO_ASSOC Open a file associated with an ADAM parameter.

FIO_CANCL Close a file and cancel the parameter.

RIO_ANNUL Annul a file descriptor and close the file.

RIO_ASSOC Open a file associated with an ADAM parameter.

RIO_CANCL Close a file and cancel the parameter.

These routines are typically used to get the name of a file through the ADAM parameter system. For instance, the previous example could be re-written as:

```
      ...
*  Open a file.
      CALL FIO_ASSOC( PNAME, 'WRITE', 'LIST', 0, FD, STATUS )
*  Write the data.
         DO I = 1, N
            CALL FIO_WRITE( FD, BUF( I ), STATUS )
         END DO
*  Close the file.
      CALL FIO_CANCL( PNAME, STATUS )
      ...
```

When the call to FIO_ASSOC is executed, the name of the file will be obtained via the parameter system. This may involve prompting the user, but the file name could equally well be defaulted from the interface file. The interface file might contain something like this:

```
PARAMETER  FILE
 TYPE     'FILENAME'
 VPATH    'PROMPT'
 PROMPT   'Name of file to be created'
 PPATH    'CURRENT,DEFAULT'
 DEFAULT  newfile.dat
END PARAMETER
```

N.B. At present, if you specify a file name that contains a directory name in an interface file, then you must use the appropriate (Unix or VMS) syntax. In the future, FIO may be enhanced to handle environment variables and logical names as part of the file specification.

4 Access Mode, Format and Record Size of Files

When a file is opened by one of FIO_OPEN, FIO_ASSOC, RIO_OPEN or RIO_ASSOC, then various attributes of the file need to be specified. These are the access mode, the format and the record size.

The access mode can be one of 'READ', 'WRITE', 'UPDATE' or 'APPEND'. 'READ' specifies that the file is to be opened for reading only. This is required if the protection of the file forbids writing to it, but it is good practice to always use this option for files that will only ever be read. 'WRITE' specifies that a new file is created and the file is opened for writing to. This also allows the file to be read, as once a record has been written, it can then be read. 'UPDATE' access opens an existing file for read and write access. 'APPEND' opens an existing file for read and write access. Any records written to the file will be added to the end of the file current file. If the file does not exist, it will be created.

The format specifies the type of the file. It can be one of 'LIST', 'FORTRAN' or 'NONE' (for FIO_OPEN and FIO_ASSOC), 'FORMATTED' (for RIO_OPEN and RIO_ASSOC), or 'UNFORMATTED'. 'LIST' specifies that the first character in a record should not be interpreted as a carriage control character, and is usually what is needed to produce simple text files. 'FORTRAN' specifies that the first character in a record will be interpreted as a carriage control character. This may be useful when producing reports that are to be printed on a line printer. The FORTRAN 77 standard says that output record that are to be printed will have their first characters interpreted as carriage control characters, and implies, but does not state explicitly, that output records that are not be be printed will not have their first characters interpreted as carriage control characters. Unfortunately, it is rather vague as to what the term *printing* actually means. An additional source of confusion is that a standard FORTRAN OPEN statement will create files that do cause the first character of each record to be interpreted as a carriage control character on VMS, but not on Unix. In fact, Unix has no concept of the type of a file, so files that have carriage control characters in them need to be passed through a filter (often called fpr) for the carriage control characters to have their desired effect.

A format of 'NONE' specifies that there is no carriage control character. This differs from a format of 'LIST' on VMS or Ultrix as the file will print on a single line when listed on a terminal or printed on a printer. On SunOS, a format of 'NONE' has the same effect as 'LIST'. It is best to avoid this option whenever possible. In fact, for formatted, sequential access files (i.e. simple text files), it is best to use a format of 'LIST' whenever possible.

A format of 'FORMATTED' will produce a formatted direct access file with RIO_OPEN or RIO_ASSOC and a format of 'UNFORMATTED' will always produce an unformatted file.

The record size is generally only needed for direct access files created by RIO_OPEN and RIO_ASSOC. In other cases it should be specified as zero, which causes FIO to use the default size of a record. In fact it is a violation of the FORTRAN 77 standard to give a record length when opening a sequential file. However, VMS requires the record length to be given when creating records that are longer than the default of 133 bytes. If a record length is given to an FIO routine on Unix, it will ignore it.

5 INCLUDE files

The include file FIO_PAR defines symbolic names for various constants which may be required by tasks. The most useful constants are FIO__SZMOD, which is used to specify the length of the access mode string in calls to FIO_OPEN and RIO_OPEN, and FIO__SZFNM, which is the maximum allowed length of a filename in FIO/RIO.

If you need to test for explicit status values returned from FIO/RIO subroutines, include the statement:

```
INCLUDE 'FIO\_ERR'
```

in the program. The return status can then be tested. For example:

```
IF( STATUS .EQ. FIO__ERROR ) ...
```

However, there are problems to do with portability when testing return status values. These are dealt with in the next section.

6 Reporting and handling errors

FIO/RIO routines all report errors if they return bad status values, so programs that do all I/O through calls to FIO/RIO do not have to worry about this. However, some programs use direct FORTRAN statements to perform I/O and may still need to report errors. Two routines are provided to assist with this; FIO_SERR and FIO_REP. FIO_SERR is the simpler of the two. It takes an IOSTAT value as its first argument, returns a corresponding FIO error value in its status argument and reports an error. The error report is of the form:

```
FIO_SERR: IOSTAT error = Unit not connected
```

This is fine if all you want to do is translate the IOSTAT value, but the error report does not contain any contextual information such as the unit that was not connected nor the file that it should have been connected to. For a fuller report, the routine FIO_REP is provided. This takes as input arguments the unit number, the file name, the IOSTAT value and a message to be printed. FIO_REP sets three message tokens, FNAME, UNIT and IOSTAT and then reports the message that it was given. This message can contain references to the message tokens to provide a more meaningful error message. For example:

```
OPEN( UNIT=UNUM, FILE=FILNAM, STATUS='OLD', IOSTAT=ISTAT )
CALL FIO_REP( UNUM, FILNAM, ISTAT,
:  'Error opening file ^FNAME. Status = ^IOSTAT', STATUS )
```

In this case, the error report contains the fact that this error has been generated when trying to open a file. To save having to generate an error message for every call to FIO_REP, it is possible to give a blank message, which is equivalent to

```
'Error with file ^FNAME on unit number ^UNIT; IOSTAT = ^IOSTAT'
```

For a given value of IOSTAT, the value of status that is returned by FIO_REP is the same as that returned by FIO_SERR.

6.1 Handling errors

Sometimes it is desired to take corrective action if a routine returns a particular bad status value, and section 5 contains an example of how you might do this. Unfortunately there is a problem with testing FIO/RIO status values that does not occur with most other packages.

FIO/RIO can generate two sorts of error codes. Firstly there are internal FIO/RIO codes. There is no problem testing for these. Secondly there are codes that are a translation of a FORTRAN IOSTAT value. It is these status codes that gives rise to the problem as such values are inherently machine specific, thus making it very difficult to write portable applications that test for bad status values. It might be thought that the things that could go wrong with FORTRAN I/O were sufficiently similar from one machine to another, that a common set of error codes could be devised, but surprisingly this is not the case in practice. The list of error codes that can be returned as IOSTAT values are very different from one machine to another. Even when it looks like two errors on different machines will be equivalent in practice, this does not always turn out to be the case.

On account of these difficulties, FIO/RIO adopts the following strategy:

> *If the text of an error message in the computer manufacturer's documentation is the same for two different machines, then FIO/RIO will return the same status value on those two machines. Otherwise different status values are returned on the different machines.*

This strategy is applied quite rigorously, even when, at first sight, it looks like two error messages might be equivalent. The only exception at present is that 'Cannot stat file' (on Ultrix) and 'can't stat file' (on SunOS) return the same error code. Not to do so smacks of pedantry of the highest order! This strategy has been chosen as a balance between returning unique error codes on all machines (which is barely any better than using the raw IOSTAT value) and trying to guess which error codes are equivalent to each other (with the likelihood of getting it wrong). Presumably if the text of two error messages are identical, then they are intended to apply to the same situation. Even this cannot be guaranteed, but it is the best one can do.

Occasionally, the Fortran run time system will return an IOSTAT value that corresponds to a operating system error rather than a Fortran error. In such a case, an error message describing the error will be generated and the status will be set to the value of the symbolic constant FIO__OSERR.

The strategy of only returning the same error number when the text of the message is the same definitely errs on the side of caution. It means that programs that are intended to be run on several different machines must often test for different error codes, one for each machine type. For example, it is quite common to test for `FIO__FILNF` (file not found) on VAX/VMS. Unfortunately, there is no error that corresponds sufficiently closely to this on SunOS. As well as being very tedious, it means that tests for bad status values in application programs probably need to be modified to run on a new computer. To minimize this problem, FIO/RIO provides the

ability to test status values for classes of errors. This is best described by an example. Suppose that you have prompted a user for the name of an input file and you then try to open a file using the returned string. If the program fails to open the file, this might be for one of several reasons. It may be that the file does not exist, or that the file exists, but you do not have the right to access the file, or that the string typed in is not a valid file name (e.g. `[PMA}TEST.DAT` on VMS). In all of these situations, you can rely on the error reporting to tell the user what has gone wrong, but all the program cares about is that it has failed to open the file and that it should re-prompt the user. A program can test for a general class of errors by using the logical function FIO_TEST. This takes a character argument and a status value and returns TRUE if the value of STATUS is in the class of errors described by the character argument. Here is an example:

```
IF( FIO_TEST( 'OPEN error', STATUS ) ) THEN
   ...
ENDIF
```

Note that FIO_TEST is not sensitive to the case of the character string given as its first argument. An example of attempting to open a file using FORTRAN I/O and then testing to see if this was successful is:

```
      CALL ERR_MARK
      OPEN( UNIT=UNUM, FILE=FILNAM, STATUS='OLD', IOSTAT=ISTAT )
      CALL FIO_REP( UNUM, FILNAM, ISTAT, ' ', STATUS )
*  Test for 'could not open file'.
      IF( FIO_TEST( 'OPEN error', STATUS ) ) THEN
*  Handle the error if we can.
         ...
         CALL ERR_ANNUL( STATUS )
      END IF
      CALL ERR_RLSE
```

This example has used a FORTRAN OPEN statement in the application code. It is generally better to let FIO handle all file access as this makes for more portable code. (It is also less typing.) In this case, the above example would be written as:

```
      CALL ERR_MARK
      CALL FIO_OPEN( FILNAM, 'UPDATE', 'LIST', 0, FD, STATUS )
      IF( FIO_TEST( 'OPEN error', STATUS ) ) THEN
*  Handle the error if we can.
         ...
         CALL ERR_ANNUL( STATUS )
      END IF
      CALL ERR_RLSE
```

A list of all the classes of errors that can be handled in this manner is given in appendix E. At present, the list of error classes is fixed, but it is intended that users will be able to define their own error classes in a future release of FIO/RIO.

6.2 Note to software developers

The routines that provide the interface to the ADAM parameter system report errors by calling the ERR library. All other routines report errors by calling the EMS library.

7 Compiling and Linking

7.1 Unix

On a Unix system, the FORTRAN compiler will only look for include files in the directory that contains the source code of the program being compiled unless the include file is given as an explicit path name. Consequently, the best way of naming include files on a Unix system is to use soft links. For example, the program contains lines such as:

```
INCLUDE 'SAE_PAR'
```

and you create a soft link in your directory with the command:

```
% ln -s /star/include/sae_par SAE_PAR
```

A shell script called `fio_dev` is provided to create the appropriate soft links for the FIO library.

To compile and link a program that uses FIO, type:

```
% f77 prog.f -L/star/lib `fio_link`
```

To compile and link an ADAM program that uses FIO, type:

```
% alink prog.f `fio_link_adam`
```

7.2 VMS

The current version of FIO/RIO is distributed as a shareable image. Before compiling a program that uses any of the FIO include files, or linking any program that uses FIO, type

```
$ FIO_DEV
```

The FIO shareable image is included in the STAR_LINK shareable image library, so the preferred method of linking basic FORTRAN programs is:

```
$ LINK progname,STAR_LINK/OPT
```

To link an ADAM program with FIO, type:

```
$ ALINK progname
```

The shareable libraries and object libraries are stored in FIO_DIR, so if you need to link explicitly with the shareable library, type:

```
$ LINK progname,FIO_LINK/OPT
```

or to link with the object library, type:

```
$ LINK progname,FIO\_DIR:FIO/LIB
```

Linking with the object library is not recommended as it makes the size of executable files larger than using shareable libraries and it will require relinking programs to take advantage of bug fixes or updates.

A Alphabetical List of Routines

FIO_ACTIV
Initialise FIO library for ADAM application

FIO_ANNUL
Annul a file descriptor and close the file

FIO_ASSOC
Create/open a sequential file associated with a parameter

FIO_CANCL
Close a file and cancel the parameter

FIO_CLOSE
Close a sequential file

FIO_DEACT
Deactivate FIO

FIO_ERASE
Delete a file

FIO_FNAME
Get the full file name of a file

FIO_GUNIT
Get a unit number

FIO_OPEN
Create/open a sequential file

FIO_PUNIT
Release a unit number

FIO_READ
Read sequential record

FIO_READF
Fast read sequential record

FIO_REP
Report error from FORTRAN I/O statements

FIO_RWIND
Rewind a sequential file

FIO_SERR
Set error status

FIO_START
Set up units numbers and open standard I/O streams

FIO_STOP
Close down FIO

FIO_TEST
Test if an FIO status value belongs to a certain class of errors

FIO_UNIT
Get a unit number given a file descriptor

FIO_WRITE
Write a sequential record

RIO_ANNUL
Annul a file descriptor and close the file

RIO_ASSOC
Create/open a direct access file associated with a parameter

RIO_CANCL
Close a file and cancel the parameter

RIO_CLOSE
Close a direct access file

RIO_ERASE
Delete a file

RIO_OPEN
Open a direct access file

RIO_READ
Read record from direct access file

RIO_WRITE
Write a record to a direct access file

B Classified List of Routines

B.1 Simple I/O routines

FIO_ERASE
Delete a file

FIO_GUNIT
Get a unit number

FIO_PUNIT
Release a unit number

FIO_REP
Report error from FORTRAN I/O statements

FIO_SERR
Set error status

FIO_TEST
Test if an FIO status value belongs to a certain class of errors

RIO_ERASE
Delete a file

B.2 Stand alone routines

FIO_CLOSE
Close a sequential file

FIO_FNAME
Get the full file name of a file

FIO_OPEN
Create/open a sequential file

FIO_READ
Read sequential record

FIO_READF
Fast read sequential record

FIO_RWIND
Rewind a sequential file

FIO_UNIT
Get a unit number given a file descriptor

FIO_WRITE
Write a sequential record

RIO_CLOSE
Close a direct access file

RIO_OPEN
Open a direct access file

RIO_READ
Read record from direct access file

RIO_WRITE
Write a record to a direct access file

B.3 ADAM parameter system routines

FIO_ANNUL
Annul a file descriptor and close the file

FIO_ASSOC
Create/open a sequential file associated with a parameter

FIO_CANCL
Close a file and cancel the parameter

RIO_ANNUL
Annul a file descriptor and close the file

RIO_ASSOC
Create/open a direct access file associated with a parameter

RIO_CANCL
Close a file and cancel the parameter

B.4 Miscellaneous routines

FIO_ACTIV
Initialise FIO library for ADAM application

FIO_DEACT
Deactivate FIO

FIO_START
Set up units numbers and open standard I/O streams

FIO_STOP
Close down FIO

C Routine Descriptions

FIO_ANNUL
Annul a file descriptor and close the file

Description:

This routine closes the file associated with the file descriptor FD, resets the file descriptor and removes the association with the ADAM parameter. It does not cancel the ADAM parameter though. This allows the value of the ADAM parameter to be reused.

Invocation:

```
CALL FIO_ANNUL( FD, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

FIO_ASSOC
Create/open a sequential file associated with a parameter

Description:

Open the sequential file specified by parameter PNAME and return a file descriptor for it.

Invocation:

```
CALL FIO_ASSOC( PNAME, ACMODE, FORM, RECSZ, FD, STATUS )
```

Arguments:

PNAME = CHARACTER * (*) (Given)

Expression giving the name of a file parameter.

ACMODE = CHARACTER * (*) (Given)

Expression giving the required access mode. Valid modes are:
'READ' - Open the file READONLY. The file must exist.
'WRITE' - Create a new file and open it to write.
'UPDATE' - Open a file to write. The file must exist.
'APPEND' - Open a file to append. The file need not exist.

FORM = CHARACTER * (*) (Given)

Expression giving the required formatting of the file. Valid formats are:
'FORTRAN' - Formatted file, normal Fortran interpretation of the first character of each record.
'LIST' - Formatted file, single spacing between records.
'NONE' - Formatted file, no implied carriage control.
'UNFORMATTED' - Unformatted, no implied carriage control.

RECSZ = INTEGER (Given)

Expression giving the maximum record size in bytes. Set it to zero if the Fortran default is required.

FD = INTEGER (Returned)

Variable to contain the file descriptor.

STATUS = INTEGER (Given and Returned)

The global status.

External Routines Used :

CHR: CHR_SIMLR

FIO_CANCL
Close a file and cancel the parameter

Description:

Close any open file that is associated with the parameter and cancel the parameter.

Invocation:

```
CALL FIO_CANCL( PNAME, STATUS )
```

Arguments:

PNAME = CHARACTER * (*) (Given)

Expression giving the name of a file parameter which has previously been associated with a file using FIO_ASSOC.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

FIO_CLOSE
Close a sequential file

Description:
Close the file with the specified file descriptor.

Invocation:

```
CALL FIO_CLOSE( FD, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the STATUS variable is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

FIO_ERASE
Delete a file

Description:

Delete the named file.

Invocation:

```
CALL FIO_ERASE( FILE, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)

Expression giving the name of the file to be deleted.

STATUS = INTEGER (Given and Returned)

The global status.

FIO_FNAME
Get the full file name of a file

Description:

Get the full name of the file with the specified file descriptor.

Invocation:

```
CALL FIO_FNAME( FD, FNAME, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

FNAME = CHARACTER * (*) (Returned)
Variable to contain the full file name of the file.

STATUS = INTEGER (Given and Returned)
The global status.

FIO_GUNIT
Get a unit number

Description:

Get an unused Fortran unit number.

Invocation:

```
CALL FIO_GUNIT( UNIT, STATUS )
```

Arguments:

UNIT = INTEGER (Given)

A variable to contain the unit number.

STATUS = INTEGER (Given and Returned)

The global status.

FIO_OPEN
Create/open a sequential file

Description:

Open a sequential file with the specified access mode. When the file is created, the specified carriage control mode and maximum record size will be used. Return a file descriptor which can be used to access the file.

Invocation:

```
CALL FIO_OPEN( FILE, ACMODE, FORM, RECSZ, FD, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)

Expression giving the name of the file to be opened.

ACMODE = CHARACTER * (*) (Given)

Expression giving the required access mode. Valid modes are:
'READ' - Open the file READONLY. The file must exist.
'WRITE' - Create a new file and open it to write.
'UPDATE' - Open a file to write. The file must exist.
'APPEND' - Open a file to append. The file need not exist.

FORM = CHARACTER * (*) (Given)

Expression giving the required formatting of the file. Valid formats are:
'FORTRAN' - Formatted file, normal Fortran interpretation of the first character of each record.
'LIST' - Formatted file, single spacing between records.
'NONE' - Formatted file, no implied carriage control.
'UNFORMATTED' - Unformatted, no implied carriage control.

RECSZ = INTEGER (Given)

Expression giving the maximum record size in bytes. Set it to zero if the Fortran default is required.

FD = INTEGER (Returned)

Variable to contain the file descriptor.

STATUS = INTEGER (Given and Returned)

The global status.

FIO_PUNIT
Release a unit number

Description:

Give back a Fortran unit number to FIO.

Invocation:

```
CALL FIO_PUNIT( UNIT, STATUS )
```

Arguments:

UNIT = INTEGER (Given)

Variable containing the unit number.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

If STATUS is not set to SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

FIO_READ
Read sequential record

Description:

Read a record from the file with the specified file descriptor and return the 'used length' of the buffer.

Invocation:

```
CALL FIO_READ( FD, BUF, NCHAR, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

BUF = CHARACTER * (*) (Returned)
Variable to receive the record.

NCHAR = INTEGER (Returned)
Variable to receive the number of characters read, ignoring trailing spaces.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

FIO_READ reflects the behaviour of the underlying Fortran I/O system so identical behaviour on different platforms cannot be guaranteed. In particular, platforms differ in the way they handle records which are terminated by EOF rather than newline. Supported platforms currently behave as follows:

	Buffer	STATUS	NCHAR
Alpha:	Trailing spaces added	SAI__OK	Used length
Solaris:	No trailing spaces added	FIO__EOF	0
Linux:	No trailing spaces added	FIO__EOF	0

In the interests of efficiency, the buffer is not cleared before each READ so it is not possible for FIO_READ to find the used length on Solaris or Linux in this case. The programmer may do so if required.

External Routines Used :

CHR: CHR_LEN

FIO_READF
Fast read sequential record

Description:

Read a record from the file with the specified file descriptor. Unlike FIO_READ, this routine does not return the 'used length' of the buffer and is therefore faster.

Invocation:

```
CALL FIO_READF( FD, BUF, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

BUF = CHARACTER * (*) (Returned)
Variable to receive the record.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

FIO_READF reflects the behaviour of the underlying Fortran I/O system so identical behaviour on different platforms cannot be guaranteed. In particular, platforms differ in the way they handle records which are terminated by EOF rather than newline. Supported platforms currently behave as follows:

	Buffer	STATUS
Alpha:	Trailing spaces added	SAI__OK
Solaris:	No trailing spaces added	FIO__EOF
Linux:	No trailing spaces added	FIO__EOF

FIO_REP
Report error from FORTRAN I/O statements

Description:

Translate the value of IOSTAT to an FIO error code and report the corresponding error message.

Invocation:

```
CALL FIO_REP( UNIT, FNAME, IOSTAT, MESS, STATUS )
```

Arguments:

UNIT = INTEGER (Given)
The Fortran I/O unit number.

FNAME = CHARACTER * (*) (Given)
The name of the data file.

IOSTAT = INTEGER (Given)
The value of IOSTAT from a Fortran I/O statement.

MESS = CHARACTER * (*) (Given)
An error message to be output.

STATUS = INTEGER (Given and Returned)
The global status.

Examples:

```
CALL FIO_REP( UNIT, ' ', IOSTAT, ' ', STATUS )
```

This will inquire the name of the file that is connected to UNIT and report an error message containing the unit number file name and which error occurred.

```
CALL FIO_REP( UNIT, ' ', IOSTAT, 'Failed to open ^FNAME', STATUS )
```

This example provides an explicit error message containing the token FNAME.

Notes:

- This routine sets the message tokens UNIT, FNAME and IOSTAT. They can be given in the text of the error message.
- FNAME can be a general character string, a hyphen or blank. If FNAME is a general character string, it is used as the name of the file when reporting the error message. If FNAME is blank, then this routine uses INQUIRE to find the name of the file. If FNAME is a hyphen, then this routine does not set the token FNAME. It should be set before calling this routine if a sensible error message is to be produced.

FIO_RWIND
Rewind a sequential file

Description:
Rewind a sequential access file.

Invocation:

```
CALL FIO_RWIND( FD, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine must ONLY be used on sequential access files.

Bugs:
None known.

FIO_SERR
Set error status

Description:

Convert a Fortran IOSTAT error value into an FIO status value and report the error.

Invocation:

```
CALL FIO_SERR( IOSTAT, STATUS )
```

Arguments:

IOSTAT = INTEGER (Given)

Variable containing the Fortran error value.

STATUS = INTEGER (Given and Returned)

The global status. Set to contain the FIO status.

FIO_TEST
Test if an FIO status value belongs to a certain class of errors

Description:

See if the value of STATUS corresponds one of the FIO error codes that correspond to the error class given as the first argument.

Invocation:

```
RESULT = FIO_TEST( ERRCLS, STATUS )
```

Arguments:

ERRCLS = CHARACTER * (*) (Given)
The name of the error class

STATUS = INTEGER (Given and Returned)
The global status.

Returned Value:

FIO_TEST = LOGICAL
Whether STATUS is in the named class of errors.

Examples:

```
IF( FIO_TEST( 'OPEN ERROR', STATUS ) ) THEN ...
```

See if the value of STATUS is one of the values associated with the error class 'OPEN ERROR'.

External Routines Used :

CHR: CHR_SIMLR

FIO_UNIT
Get a unit number given a file descriptor

Description:

The Fortran unit number associated with the given file descriptor is returned.

Invocation:

```
CALL FIO_UNIT( FD, UNIT, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

UNIT = INTEGER (Returned)
Variable to receive the unit number.

STATUS = INTEGER (Given and Returned)
The global status.

FIO_WRITE
Write a sequential record

Description:

Write a buffer to the file with the specified file descriptor.

Invocation:

```
CALL FIO_WRITE( FD, BUF, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

BUF = CHARACTER (*) (Given)
Expression containing the data to be written.

STATUS = INTEGER (Given and Returned)
The global status.

RIO_ANNUL
Annul a file descriptor and close the file

Description:

This routine closes the file associated with the file descriptor FD, resets the file descriptor and removes the association with the ADAM parameter. It does not cancel the ADAM parameter though. This allows the value of the ADAM parameter to be reused.

Invocation:

```
CALL RIO_ANNUL( FD, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

RIO_ASSOC
Create/open a direct access file associated with a parameter

Description:

Open the direct access file specified by parameter PNAME and return a file descriptor for it.

Invocation:

```
CALL RIO_ASSOC( PNAME, ACMODE, FORM, RECSZ, FD, STATUS )
```

Arguments:

PNAME = CHARACTER * (*) (Given)

Expression giving the name of a file parameter.

ACMODE = CHARACTER * (*) (Given)

Expression giving the required access mode. Valid modes are:
'READ' - Open the file READONLY. The file must exist.
'WRITE' - Create a new file and open it to write/read.
'UPDATE' - Open a file to read/write. The file must exist.
'APPEND' - Open a file to write/read. If the file does not already exist, create it. (APPEND has no other effect for direct access)

FORM = CHARACTER * (*) (Given)

Expression giving the required record formatting. Valid options are 'FORMATTED' or 'UNFORMATTED'

RECSZ = INTEGER (Given)

Expression giving the record size in bytes. RECSZ is only used if ACMODE is 'WRITE' or 'APPEND'. If ACMODE is 'APPEND' and the file already exists, RECSZ must agree with the existing record size.

FD = INTEGER (Returned)

Variable to contain the file descriptor.

STATUS = INTEGER (Given and returned)

Global status

External Routines Used :

CHR: CHR_SIMLR

RIO_CANCL
Close a file and cancel the parameter

Description:

Close any open file that is associated with the parameter and cancel the parameter.

Invocation:

```
CALL RIO_CANCL( PNAME, STATUS )
```

Arguments:

PNAME = CHARACTER * (*) (Given)

Expression giving the name of a file parameter which has previously been associated with a file using RIO_ASSOC.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

RIO_CLOSE
Close a direct access file

Description:

Close the file with the specified file descriptor.

Invocation:

```
CALL RIO_CLOSE( FD, STATUS )
```

Arguments:

FD = INTEGER (Given)

A variable containing the file descriptor.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

If the STATUS variable is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.

RIO_ERASE
Delete a file

Description:

Delete the named file.

Invocation:

```
CALL RIO_ERASE( FILE, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)

Expression giving the name of the file to be deleted.

STATUS = INTEGER (Given and Returned)

The global status.

RIO_OPEN
Open a direct access file

Description:

Open a direct access file with the specified access mode and record size. Return a file descriptor which can be used to access the file.

Invocation:

```
CALL RIO_OPEN( FILE, ACMODE, FORM, RECSZ, FD, STATUS )
```

Arguments:

FILE = CHARACTER * (*) (Given)

Expression giving the name of the file to be opened.

ACMODE = CHARACTER * (*) (Given)

Expression giving the required access mode. Valid modes are:
'READ' - Open the file READONLY. The file must exist.
'WRITE' - Create a new file and open it to write/read.
'UPDATE' - Open a file to read/write. The file must exist.
'APPEND' - Open a file to write/read. If the file does not already exist, create it. (APPEND has no other effect for direct access)

FORM = CHARACTER * (*) (Given)

Expression giving the required record formatting. 'FORMATTED' or 'UNFORMATTED'

RECSZ = INTEGER (Given)

Expression giving the record size in bytes. RECSZ is only used if ACMODE is 'WRITE' or 'APPEND'. If ACMODE is 'APPEND' and the file already exists, RECSZ must agree with the existing record size.

FD = INTEGER (Returned)

Variable to contain the file descriptor.

STATUS = INTEGER (Given and Returned)

The global status.

RIO_READ
Read record from direct access file

Description:

Read the specified unformatted record from the file with the given file descriptor.

Invocation:

```
CALL RIO_READ( FD, RECNO, NCHAR, BUF, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

RECNO = INTEGER (Given)
Expression giving the number of the record to be read.

NCHAR = INTEGER (Given)
Expression giving the buffer size

BUF = BYTE(NCHAR) (Returned)
A byte array to receive the record.

STATUS = INTEGER (Given and Returned)
The global status.

RIO_WRITE
Write a record to a direct access file

Description:

Write the specified record number, unformatted, to the file with the specified file descriptor.

Invocation:

```
CALL RIO_WRITE( FD, RECNO, NCHAR, BUF, STATUS )
```

Arguments:

FD = INTEGER (Given)
The file descriptor.

RECNO = INTEGER (Given)
Expression giving the number of the record to be written.

NCHAR = INTEGER (Given)
Expression giving the buffer size.

BUF = BYTE(NCHAR) (given)
A byte array containing the data to be written.

STATUS = INTEGER (Given and Returned)
The global status

D Description of Miscellaneous Routines

These routines are never needed in standard programs. However, they are documented here for completeness as they have existed for several years and there may be a case for calling them in certain time-critical applications. Calling them will not speed up a program, but can move a small amount of execution time from the body of a program to its initialization phase.

FIO_ACTIV
Initialise FIO library for ADAM application

Description:

The FIO package and parameter system is initialised for the start of an executable image.

Invocation:

```
CALL FIO_ACTIV( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This routine is not normally needed in a simple program as FIO activates itself when necessary.

FIO_DEACT
Deactivate FIO

Description:

The FIO stand-alone and environment levels are de-activated for the end of an executable image.

Invocation:

```
CALL FIO_DEACT( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.
- This routine is not normally needed as FIO is closed down by normal program termination.

FIO_START
Set up units numbers and open standard I/O streams

Description:

Allocate unit numbers for use by FIO and mark them as available. Open standard input, output and error files.

Invocation:

```
CALL FIO_START( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is not normally needed in a simple program as FIO starts itself when necessary.

FIO_STOP
Close down FIO

Description:

Close the FIO file descriptor system and all associated files.

Invocation:

```
CALL FIO_STOP( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- If STATUS is not SAI__OK on input, then the routine will still attempt to execute, but will return with STATUS set to the import value.
- This routine is not normally needed in a simple program as FIO is closed down by normal program termination.

E FIO status values and error classes

This appendix lists all of the error codes and classes.

As described in section 6, FIO/RIO can return both machine independent and machine specific error codes in the STATUS argument. Portable programs should only test for the machine independent codes or test for error classes using FIO_TEST.

Note that, historically, the codes FIO__ILLAC and FIO__IVUNT have been used both as machine independent internal FIO error codes and as VMS specific error codes. This usage is retained for compatibility. It is unlikely to cause any problems, but the user should be aware of this, particular if mixing direct FORTRAN I/O operations with FIO calls that perform actual I/O.

Internal (machine independent) FIO status values:

FIO__EOF	End of file
FIO__ERROR	Error
FIO__FDNFP	File descriptor does not have an associated file parameter descriptor
FIO__ILLAC [1]	Illegal access mode
FIO__ILLFD	Illegal file descriptor
FIO__INVRL	Invalid record length
FIO__IVUNT [1]	Invalid unit number
FIO__ISACT	File parameter is active
FIO__IVACM	Invalid access mode
FIO__IVFMT	Invalid format
FIO__NOUNT	No more unit numbers available
FIO__NTOPN	File not open
FIO__OSERR	General operating system error code
FIO__TOOFD	No more available file descriptors
FIO__TOOFP	Too many file parameters
FIO__UNKPA	Parameter is not a file parameter

Error classes:

[1] See note about multiple use of this error code

Class name	STATUS values that match the class
OPEN error	FIO__FILNF, FIO__CFOLF, FIO__COEXI, FIO__NFEXI, FIO__NAMER, FIO__NODEV, FIO__OPNER, FIO__PTAFD, FIO__PERMD, FIO__ILLOP, FIO__ALOPN, FIO__TOOMF
CLOSE error	FIO__CLSER, FIO__ILLCL, FIO__INCOC
READ error	FIO__RDER, FIO__INPCN, FIO__INREQ, FIO__SYNAM, FIO__TOOMV, FIO__RUNCH, FIO__BLINP, FIO__ILSTI, FIO__IINAM
WRITE error	FIO__WRT, FIO__REWRT, FIO__OUTCN, FIO__OUTOV
REWIND error	FIO__REWER
BACKSPACE error	FIO__BACER, FIO__CNTBF

Note that references to error classes in programs are case insensitive.

DEC FORTRAN (OSF/1, Ultrix and VMS) specific FIO status values:

[1]See note about multiple use of this error code

FIO__ALOPN	File already open
FIO__BACER	BACKSPACE error
FIO__CLSER	File close error
FIO__CNTSF	Cannot stat file (Ultrix only)
FIO__COEXI	Cannot overwrite existing file (Ultrix only)
FIO__DLTER	File delete error
FIO__DUPFL	Duplicate file
FIO__ENDFL	ENDFILE error
FIO__FILNF	File not found
FIO__FINER	FIND error
FIO__FORVR	Format/variable-type mismatch
FIO__ILLAC [1]	Illegal access mode
FIO__INCKC	Inconsistent key change or duplicate key
FIO__INCOC	Inconsistent OPEN/CLOSE parameters
FIO__INCRC	Inconsistent record length
FIO__INCRG	Inconsistent file organization
FIO__INCRT	Inconsistent record type
FIO__INFOR	Infinite format loop
FIO__INPCN	Input conversion error
FIO__INREQ	Input statement requires too much data
FIO__INSVR	Insufficient virtual memory
FIO__INVMK	Invalid key match specifier for key direction
FIO__INVKY	Invalid key specification
FIO__INVRG	Invalid argument to FORTRAN Run-Time Library
FIO__INVRV	Invalid reference to variable
FIO__IVUNT [1]	Invalid unit number
FIO__KEYVL	Keyword value error in OPEN statement
FIO__LISYN	List-directed I/O syntax error
FIO__MIXFL	Mixed file access modes
FIO__NAMER	File name error
FIO__NOCRC	No current record
FIO__NODEV	No such device
FIO__OPNER	File open error
FIO__OPREQ	OPEN or DEFINE FILE required

FIO__SPLOC	Specified record locked
FIO__SYNAM	Syntax error in NAMELIST input
FIO__SYNER	Syntax error in format
FIO__TOOMV	Too many values for NAMELIST variable
FIO__TOORC	Too many records in I/O statement
FIO__UNLER	UNLOCK error
FIO__UNTNC	Unit not connected (Ultrix only)
FIO__VFVAL	Variable format expression value error
FIO__WRTER	File write error

Sun FORTRAN specific FIO status values:

FIO__BLINP	Blank logical input field (Sun Fortran 1.x only)
FIO__CFOLF	Cannot find 'OLD' file
FIO__CNTBF	Cannot backspace file
FIO__CNTSF	Can't stat file
FIO__DIONA	Direct I/O not allowed
FIO__ERFMT	Error in format
FIO__FILEO	Error in FILEOPT parameter
FIO__FIONA	Formatted I/O not allowed
FIO__IINAM	Illegal input for namelist
FIO__ILARG	Illegal argument
FIO__ILINP	Illegal logical input field (Sun Fortran 2.x only)
FIO__ILLUN	Illegal unit number
FIO__ILOPU	Illegal operation for unit
FIO__ILSTI	Incomprehensible list input
FIO__INSPE	Incompatible specifiers in open (Sun Fortran 2.x only)
FIO__NAARC	No $*$ after repeat count
FIO__NEGRC	Negative repeat count
FIO__NFEXI	'NEW' file exists
FIO__OFBOR	Off beginning of record
FIO__OFEOR	Off end of record
FIO__OOFSP	Out of free space
FIO__REQSA	Requires seek ability
FIO__RUNCH	Read unexpected character
FIO__SIONA	Sequential I/O not allowed
FIO__TOOMF	Too many file opens – no free descriptors (Sun Fortran 1.x only)
FIO__TRUNF	Truncation failed (Sun Fortran 1.x only)
FIO__UIONA	Unformatted I/O not allowed
FIO__UNKNO	Unknown system error
FIO__UNTNC	Unit not connected
FIO__UNTNO	Attempted operation on unit that is not open (Sun Fortran 1.x only)

The following FIO error status codes may be returned on machines running SunOS or Solaris. They correspond to operating system error rather than Fortran errors. This is not an exhaustive list of all possible errors. Rather they are those errors that it seemed to the author of the package

to be worth detecting.

FIO__PERMD	Permission denied
FIO__FTOOL	File to large
FIO__NSLOD	No space left of device
FIO__FNTL	File name too long
FIO__DQEXC	Disk quota exceeded

Redundant FIO status values:

These status values are no longer used by FIO. The symbolic constants are retained so that old code that may refer to them will still compile. However, any code that tests for them as a returned status value will never find these values.

FIO__CRTER	File create error
FIO__EREXH	Error establishing exit handler
FIO__ILLCL	Illegal close request
FIO__ILLOP	Illegal open request
FIO__NOTFD	File not found (superseded by FIO__FILNF)
FIO__NTSUP	Option not supported yet
FIO__OLORG	Illegal origin
FIO__REDON	File is readonly
FIO__TOMNY	Too many open files

F Implementation details

The implementation uses FORTRAN I/O and FORTRAN 77 standards are used with the following exceptions:

F.1 Alpha OSF/1

- The READONLY keyword is used when opening files for reading only. This is required under VMS to allow a user to open a file for which only read access is permitted.
- The CARRIAGECONTROL keyword is used.
- The ACCESS = APPEND keyword is available to permit the useful but non-standard facility of appending to files.
- The RECL option on the OPEN statement is allowed with sequential files.
- The keyword ORGANIZATION (= 'RELATIVE') is used when creating direct access files.
- A byte array is used as the buffer for direct access I/O.

F.2 Sun4 Solaris

- The ACCESS = APPEND keyword is available to permit the useful but non-standard facility of appending to files.

F.3 Ultrix and sunOS/4

These are no longer fully supported but the same features apply as for alpha OSF/1 and sun4 Solaris respectively.

F.4 VMS

- The READONLY keyword is used when opening files for reading only. This is required under VMS to allow a user to open a file for which only read access is permitted.
- The CARRIAGECONTROL keyword is used.
- The ACCESS = APPEND keyword is available to permit the useful but non-standard facility of appending to files.
- The RECL option on the OPEN statement is allowed with sequential files.
- Keywords BLOCKSIZE (= 11*512) and ORGANIZATION (= 'RELATIVE') are used when creating direct access files.
- A byte array is used as the buffer for direct access I/O.

Note that the VMS implementation is frozen at release 1.4.

G Changes and new features

G.1 in version 1.5

The Unix makefile etc. have been updated to version 5, and an 'END=' specifier inserted in RIO_READ to trap a problem on Solaris if the record number is beyond the end of the file.

This release also runs on Linux.

This document has been slightly revised to reduce the prominence of VMS in the descriptions (there are no changes of substance) and to facilitate the production of the hypertext version. Although the VMS implementation is now frozen, there have been no significant developments so this document still describes both Unix and VMS implementations.

G.2 in version 1.5-2

The value of the public parameter FIO__SZFNM is increased from 80 to 200.

The Linux version has been brought in line with other platforms to return status FIO__FILNF rather than the obsolete FIO__NOTFD if it cannot find a file which is supposed to exist.

Other minor changes are made to improve the consistency of behaviour on different platforms under error conditions. A note on the effect of EOF terminated records has been added to the descriptions of FIO_READ and FIO_READF in SUN/143.

The makefile has been brought up to date - amongst other things, shared libraries will now be produced on Linux.

The format of this document has been updated and early 'Changes' sections have been removed but there is no change in other sections.