MIPSpro™ Fortran 90 Programmer's I/O Guide

007–3695–006

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
1.0	May 1994 Original Printing. This document incorporates information from the *I/O User's Guide*, publication SG-3075, and the *Advanced I/O User's Guide*, publication SG-3076.
1.2	October 1994 Revised for the Programming Environment 1.2 release.
2.0	November 1995 Revised for the Programming Environment 2.0 release.
3.0	May 1997 Revised for the Programming Environment 3.0 release.
3.0.1	August 1997 Revised for the Programming Environment 3.0.1 release and the MIPSpro 7 Fortran 90 compiler release.
3.0.2	March 1998 Revised for the Programming Environment 3.0.2 release and the MIPSpro 7 Fortran 90 compiler release.
3.1	August 1998 Revised for the Programming Environment 3.1 release.
3.2	January 1999 Revised for the Programming Environment 3.2 release.
7.3	April 1999 Revised for the MIPSpro 7.3 release.
3.3	July 1999 Revised for the Programming Environment 3.3 release.
7.3.1.1.m	October 1999 Revised for the MIPSpro 7.3.1.1.m release.

006	September 2002 Revised for the MIPSpro 7.4 release. The title of this document was changed to more clearly reflect the content.

Contents

About This Guide . . . . . **xix**
Related Publications xix
Obtaining Publications xix
Conventions xix
Reader Comments xx

1. Introduction **1**
The Message System 2

2. Standard Fortran I/O **5**
Files 5
  Internal Files 5
  External Files 6
Fortran Unit Identifiers 8
Data Transfer Statements 11
  Formatted I/O 11
    Edit-directed I/O 12
    List-directed I/O 14
      Namelist I/O 15
  Unformatted I/O 16
Auxiliary I/O 16
  File Connection Statements 16
  The INQUIRE Statement 17
  File Positioning Statements 17
Multithreading and Standard Fortran I/O 18

3. Fortran I/O Extensions **19**
BUFFER IN/BUFFER OUT Routines 19
The UNIT Intrinsic 20
The LENGTH Intrinsic 20

4. Named Pipe Support **21**
Named Pipes 21
Piped I/O Example without End-of-file Detection 22
Detecting End-of-file on a Named Pipe 24
Piped I/O Example with End-of-file Detection 24

5. System and C I/O **27**
System I/O 27
Synchronous I/O 27
Asynchronous I/O 27
Unbuffered I/O 28
C I/O from Fortran 28

6. The assign Environment **31**
assign Basics 31
Open Processing 31
The assign Command 32
Related Library Routines 35
assign and Fortran I/O 36
Alternative File Names 36
File Structure Selection 38
Buffer Size Specification 39
Foreign File Format Specification 40
Direct-access I/O Tuning 40

Fortran File Truncation 40

The `assign` Environment File 41

Local `assign` 42

7. File Structures 43

Unblocked File Structure 44

`assign -s unblocked` File Processing 45

`assign -s sbin` File Processing (Not Recommended) 45

`assign -s bin` File Processing (Not Recommended) 46

`assign -s u` File Processing 46

Text File Structure 46

COS or Blocked File Structure 47

8. Buffering 51

Buffering Overview 51

Types of Buffering 52

Unbuffered I/O 53

Library Buffering 53

System Cache 53

Default Buffer Sizes 54

9. Introduction to FFIO 55

Layered I/O 55

Using Layered I/O 57

I/O Layers 58

Layered I/O Options 59

10. Using FFIO 61

FFIO on IRIX systems 61

FFIO and Common Formats 62
Reading and Writing Text Files 62
Reading and Writing Unblocked Files 63
Reading and Writing Fixed-length Records 63
Reading and Writing COS Blocked Files 63
Enhancing Performance 64
Buffer Size Considerations 64
Removing Blocking 64
The `bufa` and `cachea` Layers 65
The `cache` Layer 65

11. Foreign File Conversion 67
Conversion Overview 67
Using `fdcp` to Transfer Files 67
Data Item Conversion 68
Explicit Data Item Conversion 68
Implicit Data Item Conversion 69
Choosing a Conversion Method 73
Explicit Conversion 73
Implicit Conversion 74
Foreign Conversion Techniques 74
Workstation and IEEE Conversion 74

12. I/O Optimization 77
Overview 77
An Overview of Optimization Techniques 79
Optimizations Not Affecting Source Code 79
Optimizations That Affect Source Code 80

Optimizing I/O Speed 80
Optimizing System Requests 81
Using a Cache Layer 81
Optimizing File Structure Overhead 82
Scratch Files 82
Using Asynchronous Read-ahead and Write-behind 83
Using Simpler File Structures 84
Minimizing Data Conversions 84
Minimizing Data Copying 85
Changing Library Buffer Sizes 85
Bypassing Library Buffers 85
Other Optimization Options 86
Using Pipes 86
Overlapping CPU and I/O 86

13. FFIO Layer Reference 89

Characteristics of Layers 89
Individual Layers 91
The `bufa` Layer 92
The `cache` Layer 93
The `cachea` Layer 95
The `cos` Blocking Layer 97
The `event` Layer 99
The `f77` Layer 100
The `fd` Layer 102
The `global` Layer 102
The `null` layer 104

The `syscall` Layer 105
The `system` Layer 106
The `text` Layer 106
The `user` and `site` Layers 108
The `vms` Layer 109

14. Creating a `user` Layer 113
Internal Functions 113
The Operations Structure 114
FFIO and the Stat Structure 115
`user` Layer Example 116

Glossary 143

Index 145

Figures

Figure 6-1 Access methods and default buffer size (IRIX systems) 41
Figure 9-1 Typical data flow 55
Figure 12-1 I/O layers 78

Tables

Table 7-1 Fortran access methods and options 44
Table 9-1 Available I/O Layers 58
Table 11-1 Available conversion routines 69
Table 11-2 Conversion types 70
Table 11-3 Supported foreign I/O formats and default data types 71
Table 13-1 Data manipulation: `bufa` layer 92
Table 13-2 Supported operations: `bufa` layer 92
Table 13-3 Data manipulation: `cache` layer 94
Table 13-4 Supported operations: `cache` layer 94
Table 13-5 Data manipulation: `cachea` layer 96
Table 13-6 Supported operations: `cachea` layer 96
Table 13-7 Data manipulation: `cos` layer 98
Table 13-8 Supported operations: `cos` layer 98
Table 13-9 Data manipulation: `f77` layer 101
Table 13-10 Supported operations: `f77` layer 101
Table 13-11 Data manipulation: `global` layer 103
Table 13-12 Supported operations: `global` layer 103
Table 13-13 Data manipulation: `syscall` layer 105
Table 13-14 Supported operations: `syscall` layer 105
Table 13-15 Data manipulation: `text` layer 107
Table 13-16 Supported operations: `text` layer 107
Table 13-17 Values for record size: `vms` layer 110
Table 13-18 Values for maximum block size: `vms` layer 110

Table 13-19 Data manipulation: `vms` layer 111

Table 13-20 Supported operations: `vms` layer 111

Examples

Example 4-1 No EOF detection: `writerd` 23
Example 4-2 No EOF detection: `readwt` 23
Example 4-3 EOF detection: `writerd` 25
Example 4-4 EOF detection: `readwt` 25
Example 5-1 C I/O from Fortran 29
Example 6-1 local `assign` mode 42

Procedures

Procedure 2-1 Optimization technique: using single statements 12

Procedure 2-2 Optimization technique: using longer records 13

Procedure 2-3 Optimization technique: using repeated edit descriptors 14

Procedure 2-4 Optimization technique: using data edit descriptors 14

About This Guide

This publication describes Fortran input/output (I/O) techniques. Information about the interaction of the I/O library and supported compilers is also discussed. This document also serves as an I/O optimization guide for Fortran programmers. It describes the types of I/O that are available, including insight into the efficiencies and inefficiencies of each, the ways to speed up various forms of I/O, and the tools used to extract statistics from the execution of a Fortran program.

Related Publications

The following documents contain additional information that may be helpful:

- *MIPSpro Fortran 90 Commands and Directives Reference Manual*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*
- *MIPSpro Fortran Language Reference Manual, Volume 3*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://techpubs.sgi.com.`

Conventions

The following conventions are used throughout this documentation:

`command`	This fixed-space font denotes literal items, such as pathnames, man page names, commands, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.

[ ]	Brackets enclose optional portions of a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

We value your comments and will respond to them promptly.

Introduction

This manual introduces standard Fortran, supported Fortran extensions, and provides a discussion of flexible file input/output (FFIO) and other input/output (I/O) methods. This manual is for Fortran programmers who need general I/O information or who need information on how to optimize their I/O.

This manual contains the following chapters:

- Chapter 2, "Standard Fortran I/O", page 5, discusses elements of the Fortran 95 standard that relate to I/O.
- Chapter 3, "Fortran I/O Extensions ", page 19, discusses extensions to the Fortran standard.
- Chapter 4, "Named Pipe Support ", page 21, discusses tape handling and FIFO special files.
- Chapter 5, "System and C I/O ", page 27, discusses system calls and Fortran callable entry points to C library routines.
- Chapter 6, "The `assign` Environment", page 31, discusses the use of the `assign`(1) command to access and update advisory information from the I/O library and how to create an I/O environment.
- Chapter 7, "File Structures ", page 43, discusses native file structures.
- Chapter 8, "Buffering", page 51, discusses file buffering as it applies to I/O.
- Chapter 9, "Introduction to FFIO ", page 55, provides an overview of the Flexible File I/O system.
- Chapter 10, "Using FFIO ", page 61, describes how to use FFIO with common file structures, and how to use FFIO to enhance program performance.
- Chapter 11, "Foreign File Conversion", page 67, discusses how to convert data from one file structure to another.
- Chapter 12, "I/O Optimization ", page 77, discusses methods to speed up I/O processing.
- Chapter 13, "FFIO Layer Reference ", page 89, provides details about individual FFIO layers.

- Chapter 14, "Creating a `user` Layer ", page 113, provides an example of how to create an FFIO layer.

The Message System

An error message system is provided that consists of commands, library routines, and files that allow error messages to be retrieved from message catalogs and formatted at run time.

The user who receives a message can request more information by using the `explain`(1) user command. The `explain` command retrieves a message explanation from an online explanation catalog and displays it on the standard output device.

The *msgid* argument to the `explain` command is the message ID string that appears when an error message is written. The ID string contains a product group code and the message number.

The product group code or product code is a string that identifies the product issuing the message. The product code for the Fortran libraries and for the I/O libraries is `lib`. The number specifies the number of the message. The following list describes the categories of message numbers:

- All Fortran library errors are within the range of 4000–5000. Libraries may also return system error numbers in the range of 1 to the first library error number. You must use the `sys` product code with numbers in this range.
- Flexible file I/O (FFIO) returns error values that are in the range of 5000 to 6000 and have a product code of `lib`.

Both of the following are variations of the `explain` command used with a *msgid* from the Fortran I/O library:

```
explain lib1100
```

```
explain lib-1100
```

The previous `explain` command produces the following description on a standard output file:

```
explain lib-1100
lib-1100: A READ operation tried to read a nonexistent record.

On a Fortran READ statement, the REC (record) specifier was
```

```
larger than the largest record number for that direct-access
file. Check the value of the REC specifier to ensure that it
is a valid record number. Check the file being read to ensure
that it is the  correct file. Also see the description of
input/output statements in your Fortran reference manual. The
class of the error is unrecoverable (issued by the Fortran
run-time library).
```

There are two classes of Fortran library error messages: `UNRECOVERABLE` and `WARNING`.

The following is an example of a warning message:

```
lib-1951 a.out: At line <n> in Fortran routine "<name>", in
     dimension <d>, extents <e1> and <e2> are not equal.
```

```
When bounds checking is enabled, this message is issued if an array
assignment exceeds the bounds of the result array. The line
number <n> in the Fortran routine <name> is where the two array
extents (<el> and <e2>) did not match.
Modify the program so as not exceed the bounds of the array, or
ensure that the array extents are equal.
Also see the description of array operations in your Fortran
reference manual.
Note that this message is issued as a warning. Execution of the
program will continue.
```

If the message number is not valid, a message similar to the following appears:

```
explain: no explanation for lib-3000
```

Chapter 2

Standard Fortran I/O

The Fortran standard describes program statements that you can use to transfer data between external media (external files) or between internal files and internal storage. It describes auxiliary input/output (I/O) statements that can be used to change the position in the external file or to write an endfile record. It also describes auxiliary I/O statements that describe properties of the connection to a file or that inquire about the properties of that connection.

Files

The Fortran standard specifies the form of the input data that a Fortran program processes and the form of output data resulting from a Fortran program. It does not specifically describe the physical properties of I/O records, files, and units. This section provides a general overview of files, records, and units.

Standard Fortran has two types of files: external and internal. An *external file* is any file that is associated with a unit number. An *internal file* is a character variable that is used as the unit specifier in a `READ` or `WRITE` statement. A *unit* is a means of referring to an external file. A unit is connected or linked to a file through the `OPEN` statement in standard Fortran. An external unit identifier refers to an external file and an internal file identifier refers to an internal file. See "Fortran Unit Identifiers", page 8, for more information about unit identifiers.

A file can have a name that can be specified through the `FILE=` specifier in a Fortran `OPEN` statement. If no explicit `OPEN` statement exists to connect a file to a unit, and if `assign`(1) was not used, the I/O library uses a form of the unit number as the file name.

Internal Files

Internal files provide a means of transferring and converting text stored in character variables. An internal file must be a character variable or character array. If the file is a variable, the file can contain only one record. If the file is a character array, each element within the array is a record. On output, the record is filled with blanks if the number of characters written to a record is less than the length of the record. An internal file is always positioned at the beginning of the first record prior to data transfer. Internal files can contain only formatted records.

When reading and writing to an internal file, only sequential formatted data transfer statements that do not specify list-directed formatting may be used. Only sequential formatted `READ` and `WRITE` statements may specify an internal file.

External Files

In standard Fortran, one external unit may be connected to a file. SGI allows more than one external unit to be connected to the standard input, standard output, or standard error files if the files were assigned with the `assign -D` command. More than one external unit can be connected to a terminal.

External files have properties of form, access, and position as described in the following text. You can specify these properties explicitly by using an `OPEN` statement on the file. The Fortran standard provides specific default values for these properties.

- **Form (formatted or unformatted)**: external files can contain formatted or unformatted records. Formatted records are read or written by formatted I/O data transfer statements. Unformatted records are accessed through unformatted I/O data transfer statements. If the default does not match the form needed, you can specify the form by using an `OPEN` statement.
- **File access (sequential or direct access)**: external files can be accessed through sequential or direct access methods. The file access method is determined when the file is connected to a unit.
  - Sequential access does not require an explicit open of a file by using an `OPEN` statement.

    When connected for sequential access, the external file has the following properties:

    - The records of the file are either all formatted or unformatted, except that the last record of the file may be an endfile record.
    - The records of the file must not be read or written by direct-access I/O statements when the file is opened for sequential access.
    - If the file is created with sequential access, the records are stored in the order in which they are written (that is, sequentially).

    To use sequential access on a file that was created as a formatted direct-access file, open the file as sequential. To use sequential access on a file that was created as an unformatted direct-access file, open the file as sequential, and use the `assign` command on the file as follows:

```
assign -s unblocked ...
```

The `assign` command is required to specify the type of file structure. The I/O libraries need this information to access the file correctly.

Buffer I/O files are unformatted sequential access files.

– Direct access does require an explicit open of a file by using an `OPEN` statement. If a file is accessed through a sequential access `READ` or `WRITE` statement, the I/O library implicitly opens the file. During an explicit or implicit open of a file, the I/O library tries to access information generated by the `assign`(1) command for the file.

  Direct access can be faster than sequential access when a program must access a set of records in a nonsequential manner.

  When connected for direct access, an external file has the following properties:

  - The records of the file are either all formatted or all unformatted. If the file can be accessed as a sequential file, the endfile record is not considered part of the file when it is connected for direct access. Some sequential files do not contain a physical endfile record.
  - The records of the file must not be read or written by sequential-access I/O statements while the file is opened for direct access.
  - All records of the file have the same length, which is specified in the `RECL` specifier of the `OPEN` statement.
  - Records do not have to be read or written in the order of their record numbers.
  - The records of the file must not be read or written using list-directed or namelist formatting.
  - The record number (a positive integer) uniquely identifies each record.

  If all of the records in the file are the same length and if the file is opened as direct access, a formatted sequential-access file can be accessed as a formatted direct-access file if the direct access file is assigned a `text` structure (with `assign -s text`).

Unformatted sequential-access files can be accessed as unformatted direct-access files if all of the records are the same length and if the file is opened as direct access, but only if the sequential-access file was created with an `unblocked` file structure. The following `assign` commands create these file structures:

```
assign -s unblocked ...
assign -s u ...
assign -F system ...
```

For more information about the `assign` environment and about default file structures, see Chapter 6, "The `assign` Environment", page 31.

- **File position**: a file connected to a unit has a *position property*, which can be either an initial point or a terminal point. The *initial point* of a file is the position just before the first record, and the *terminal point* is the position just after the last record. If a file is positioned within a record, that record is considered to be the current record; otherwise, there is no current record.

  During an I/O data transfer statement, the file can be positioned within a record as each individual input/output list (*iolist*) item is processed. The use of a dollar sign (`$`) or a backslash (\) as a carriage control edit descriptor in a format may cause a file to be positioned within a record.

  In standard Fortran, the end-of-file (EOF) record is a special record in a sequential access file; it denotes the last record of a file. A file can be positioned after an EOF, but only `CLOSE`, `BACKSPACE`, or `REWIND` statements are then allowed on the file in standard Fortran. Other I/O operations are allowed after an EOF to provide multiple-file I/O if a file is assigned to certain devices or is assigned with a certain file structure.

Fortran Unit Identifiers

A Fortran unit identifier is required for Fortran `READ` or `WRITE` statements to uniquely identify the file. A unit identifier can be one of the following:

- An integer variable or expression whose value is greater than or equal to 0. Each integer unit identifier *i* is associated with the `fort.`*i* file, which may exist (except as noted in the following text). For example, unit 10 is associated with the `fort.10` file in the current directory.

- An asterisk (`*`) is allowed only on `READ` and `WRITE` statements. It identifies a particular file that is connected for formatted, sequential access. On `READ` statements, an asterisk refers to unit 100 (standard input). On `WRITE` statements, an asterisk refers to unit 101 (standard output).

Certain Fortran I/O statements have an implied unit number. The `PRINT` statement always refers to unit 101 (standard output), and the outmoded `PUNCH` statement always refers to unit 102 (standard error).

Fortran `INQUIRE` and `CLOSE` statements may refer to any valid or invalid unit number (if referring to an invalid unit number, no error is returned). All other Fortran I/O statements may refer only to valid unit numbers. For the purposes of an executing Fortran program, all unit numbers in use or available for use by that program are valid; that is, they exist. All unit numbers not available for use are not valid; that is, they do not exist.

Valid unit numbers are all nonnegative numbers except 100 through 102. Unit numbers 0, 5, and 6 are associated with the standard error, standard input, and standard output files; any unit can also refer to a pipe. All other valid unit numbers are associated with the `fort.`*i* file, or with the file name implied in a Hollerith unit number. Use the `INQUIRE` statement to check the validity (existence) of any unit number prior to using it, as in the following example:

```
logical UNITOK, UNITOP...
inquire (unit=I,exist=UNITOK,opened=UNITOP)
if (UNITOK .and. .not. UNITOP) then
  open (unit = I, ...)
endif
```

All valid units are initially closed. A unit is connected to a file as the result of one of three methods of opening a file or a unit:

- An *implicit open* occurs when the first reference to a unit number is an I/O statement other than `OPEN`, `CLOSE`, `INQUIRE`, `BACKSPACE`, `ENDFILE`, or `REWIND`. The following example shows an implicit open:

  ```
  WRITE (4) I,J,K
  ```

  If unit number 4 is not open, the `WRITE` statement causes it to be connected to the associated file `fort.4`, unless overridden by an `assign` command that references unit 4.

  The `BACKSPACE`, `ENDFILE`, and `REWIND` statements do not perform an implicit `OPEN`. If the unit is not connected to a file, the requested operation is ignored.

- An *explicit unnamed open* occurs when the first reference to a unit number is an OPEN statement without a FILE specifier. The following example shows an explicit unnamed open:

  ```
  OPEN (7, FORM='UNFORMATTED')
  ```

  If unit number 7 is not open, the OPEN statement causes it to be connected to the associated file fort.7, unless an assign(1) command that references unit 7 overrides the default file name.

- An *explicit named open* occurs when the first reference to a unit number is an OPEN statement with a FILE specifier. The following is an example:

  ```
  OPEN (9, FILE='blue')
  ```

  If unit number 9 is not open, the OPEN statement causes it to be connected to file blue, unless overridden by an assign command that references the file named blue.

Unit numbers 100, 101, and 102 are permanently associated with the standard input, standard output, and standard error files, respectively. These files can be referenced on READ and WRITE statements. A CLOSE statement on these unit numbers has no effect. An INQUIRE statement on these unit numbers indicates they are nonexistent (not valid).

These unit numbers exist to allow guaranteed access to the standard input, standard output, and standard error files without regard to any unit actions taken by an executing program. Thus, a READ or WRITE I/O statement with an asterisk unit identifier (which is equivalent to unit 101) or a PRINT statement always works. Nonstandard I/O operations such as BUFFER IN and BUFFER OUT, READMS, and WRITMS on these units are not supported.

Fortran applications or library subroutines that must access the standard input, standard output, and standard error files can be certain of access by using unit numbers 100 through 102, even if the user program closes or reuses unit numbers 0, 5, and 6.

For all unit numbers associated with the standard input, standard output, and standard error files, the access mode and form must be sequential and formatted. The standard input file is read only, and the standard output and standard error files are write only. REWIND and BACKSPACE statements are permitted on these files but have no effect. ENDFILE statements are permitted on terminal files unless they are read only. The ENDFILE statement writes a logical endfile record.

The `REWIND` statement is not valid for any unit numbers associated with pipes. The `BACKSPACE` statement is not valid if the device on which the file exists does not support repositioning. `BACKSPACE` after a logical endfile record does not require repositioning because the endfile record is only a logical representation of an endfile record.

Data Transfer Statements

The `READ` statement is the data transfer input statement. The `WRITE` and `PRINT` statements are the data transfer output statements. If the data transfer statement contains a format specifier, the data transfer statement is a formatted I/O statement. If the data transfer statement does not contain a format specifier, the data transfer statement is an unformatted I/O statement. The time required to convert input or output data to the proper form adds to the execution time for formatted I/O statements. Unformatted I/O maintains binary representations of the data. Very little CPU time is required for unformatted I/O compared to formatted I/O.

Formatted I/O

In formatted I/O, data is transferred with editing. Formatted I/O can be edit-directed, list-directed, and namelist I/O. If the format identifier is an asterisk, the I/O statement is a list-directed I/O statement. All other format identifiers indicate edit-directed I/O.

Formatted I/O should be avoided when I/O performance is important. Unformatted I/O is faster and it avoids potential inaccuracies due to conversion. However, there are occasions when formatted I/O is necessary. The advantages for formatted I/O are as follows:

- Formatted data can be interpreted by humans.
- Formatted data can be readily used by programs and utilities not written in Fortran, or otherwise unable to process Fortran unformatted files.
- Formatted data can be readily exchanged with other computer systems where the structure of Fortran unformatted files may be different.

See the Fortran Language Reference manuals for your compiler system for more information about formatted I/O statements.

Edit-directed I/O

The format used in an edit-directed I/O statement provides information that directs the editing between internal representation and the character strings of a record (or sequence of records) in the file.

An example of a sequential access, edit-directed `WRITE` statement follows:

```
C  Sequential edit-directed WRITE statement
C
   WRITE (10,10,ERR=101,IOSTAT=IOS) 100,200
10 FORMAT (TR2,I10,1X,I10)
```

An example of a sequential access, edit-directed `READ` statement follows:

```
C  Sequential edit-directed READ statement
C
   READ (10,11,END=99,ERR=102,IOSTAT=IOS) IVAR
11 FORMAT (BN,TR2,I10:1X,I10)
```

An example of a direct access edit-directed I/O statement follows:

```
   OPEN (11,ACCESS='DIRECT',FORM='FORMATTED',
+  RECL=24)
C
C  Direct edit-directed READ and WRITE statements
C
   WRITE (11,10,REC=3,ERR=103,IOSTAT=IOS) 300,400
   READ (11,11,REC=3,ERR=104,IOSTAT=IOS) IVAR
```

There are four general optimization techniques that you can use to improve the efficiency of edit-directed formatted I/O.

Procedure 2-1 Optimization technique: using single statements

Read or write as much data with a single `READ/WRITE/PRINT` statement as possible. The following is an example of an inefficient way to code a `WRITE` statement:

```
    DO J=1,M
       DO I=1,N
         WRITE (42, 100) X(I,J)
100      FORMAT (E25.15)
       ENDDO
    ENDDO
```

It is better to write the entire array with a single `WRITE` statement, as is done in the following two examples:

```
    WRITE (42, 100) ((X(I,J),I=1,N),J=1,M)
100 FORMAT (E25.15)
```

or

```
    WRITE (42, 100) X
100 FORMAT (E25.15)
```

Each of these three code fragments produce exactly the same output; although the latter two are about twice as fast as the first. Note that the format can be used to control how much data is written per record. Also, the last two cases are equivalent if the implied `DO` loops write out the entire array, in order and without omitting any items.

Procedure 2-2 Optimization technique: using longer records

Use longer records if possible. Because a certain amount of processing is necessary to read or write each record, it is better to write a few longer records instead of more shorter records. For example, changing the statement from Example 1 to Example 2 causes the resulting file to have one fifth as many records and, more importantly, causes the program to execute faster:

Example 1: **(Not recommended)**

```
    WRITE (42, 100) X
100 FORMAT (E25.15)
```

Example 2: **(Recommended)**

```
    WRITE (42,101) X
101 FORMAT (5E25.15)
```

You must make sure that the resultant file does not contain records that are too long for the intended application. Certain text editors and utilities, for example, cannot process lines that are longer than a predetermined limit. Generally lines that are 128 characters or less are safe to use in most applications.

Procedure 2-3 Optimization technique: using repeated edit descriptors

Use repeated edit descriptors whenever possible. Instead of using the format in Example 1, use the format in Example 2 for integers which fit in four digits (that is, less than 10000 and greater than –1000).

Example 1: **(Not recommended)**

```
200 FORMAT (16(X,I4))
```

Example 2: **(Recommended)**

```
201 FORMAT (16(I5))
```

Procedure 2-4 Optimization technique: using data edit descriptors

Character data should be read and written using data edit descriptors that are the same width as the character data. For CHARACTER**n* variables, the optimal data edit descriptor is A (or A*n*). For Hollerith data in INTEGER variables, the optimal data edit descriptor is A8 (or R8).

List-directed I/O

If the format specifier is an asterisk, list-directed formatting is specified. The `REC=` specifier must not be present in the I/O statement.

In list-directed I/O, the I/O records consist of a sequence of values separated by value separators such as commas or spaces. A tab is treated as a space in list-directed input, except when it occurs in a character constant that is delimited by apostrophes or quotation marks.

List-directed and namelist output of real values uses either an `F` or an `E` format with a number of decimal digits of precision that assures full-precision printing of the real values. This allows formatted, list–directed, or namelist input of real values to result later in the generation of bit-identical binary floating point representation. Thus, a value may be written and then reread without changing the stored value.

The `LISTIO_PRECISION` and `LISTIO_OUTPUT_STYLE` environment variables can be used to control list-directed output, as discussed in the following paragraphs.

You can set the `LISTIO_PRECISION` environment variable to control the number of digits of precision printed by list-directed or namelist output. The following values can be assigned to `LISTIO_PRECISION`:

`FULL`	Prints full precision (this is the default value).
`PRECISION`	Prints *x* or *x* +1 decimal digits, where *x* is a value of the Fortran 95 `PRECISION()` intrinsic function for a given real value. This is a smaller number of digits that usually ensures that the last decimal digit is accurate to within 1 unit.

An example of a list-directed `WRITE` statement follows:

```
C  Sequential list-directed WRITE statement
   WRITE (10,*,ERR=101,IOSTAT=IOS) 100,200
```

An example of a list-directed `READ` statement follows:

```
C  Sequential list-directed READ statement
   READ (10,*,END=99,ERR=102,IOSTAT=IOS) IVAR
```

Namelist I/O

Namelist I/O is similar to list-directed I/O, but it allows you to group variables by specifying a namelist group name. On input, any namelist item within that list may appear in the input record with a value to be assigned. On output, the entire namelist is written.

The namelist item name is used in the namelist input record to indicate the namelist item to be initialized or updated. During list-directed input, the input records must contain a value or placeholder for all items in the input list. Namelist does not require that a value be present for each namelist item in the namelist group.

You can specify a namelist group name in `READ`, `WRITE`, and `PRINT` statements.

The following is an example of namelist I/O:

```
NAMELIST/GRP/T,I
READ(5,GRP)
WRITE(6,GRP)
```

Unformatted I/O

During unformatted I/O, binary data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written. The unit must be an external unit.

The following is an example of a sequential access unformatted I/O `WRITE` statement:

```
C  Sequential unformatted WRITE statement
   WRITE (10,ERR=101,IOSTAT=IOS) 100,200
```

The following is an example of a sequential access unformatted I/O `READ` statement:

```
C  Sequential unformatted READ statement
   READ (10,END=99,ERR=102,IOSTAT=IOS) IVAR
```

The following is an example of a direct access unformatted I/O statement:

```
   OPEN (11,ACCESS='DIRECT',FORM='UNFORMATTED', RECL=24)
C  Direct unformatted READ and WRITE statements
   WRITE (11,REC=3,ERR=103,IOSTAT=IOS) 300,400
   READ (11,REC=3,ERR=103,IOSTAT=IOS) IVAR
```

Auxiliary I/O

The auxiliary I/O statements consist of the `OPEN`, `CLOSE`, `INQUIRE`, `BACKSPACE`, `REWIND`, and `ENDFILE` statements. These types of statements specify file connections, describe files, or position files. See the Fortran Language Reference manual for your compiler system for more details about auxiliary I/O statements.

File Connection Statements

The `OPEN` and `CLOSE` statements specify an external file and how to access the file.

An `OPEN` statement connects an existing file to a unit, creates a file that is preconnected, creates a file and connects it to a unit, or changes certain specifiers of a connection between a file and a unit. The following are examples of the `OPEN` statement:

```
OPEN (11,ACCESS='DIRECT',FORM='FORMATTED',RECL=24)
OPEN (10,ACCESS='SEQUENTIAL', FORM='UNFORMATTED')
OPEN (9,BLANK='NULL')
```

The CLOSE statement terminates the connection of a particular file to a unit. A unit that does not exist or has no file connected to it may appear within a CLOSE statement; this would not affect any files.

The INQUIRE Statement

The INQUIRE statement describes the connection to an external file. This statement can be executed before, during, or after a file is connected to a unit. All values that the INQUIRE statement assigns are current at the time that the statement is executed.

You can use the INQUIRE statement to check the properties of a specific file or check the connection to a particular unit. The two forms of the INQUIRE statement are INQUIRE by file and INQUIRE by unit.

The INQUIRE by file statement retrieves information about the properties of a particular file.

The INQUIRE by unit statement retrieves the name of a file connected to a specified unit if the file is a named file. The standard input, standard output, and standard error files are unnamed files. An INQUIRE on a unit connected to any of these files indicates that the file is unnamed.

An INQUIRE by unit on any unit connected by using an explicit named OPEN statement indicates that the file is named, and returns the name that was present in the FILE= specifier in the OPEN statement.

An INQUIRE by unit on any unit connected by using an explicit unnamed OPEN statement, or an implicit open may indicate that the file is named. A name is returned only if the I/O library can ensure that a subsequent OPEN statement with a FILE= name will connect to the same file.

File Positioning Statements

The BACKSPACE and REWIND statements change the position of the external file. The ENDFILE statement writes the last record of the external file.

You cannot use file positioning statements on a file that is connected as a direct access file. The REC= record specifier is used for positioning in a READ or WRITE statement on a direct access file.

The BACKSPACE statement causes the file connected to the specified unit to be positioned to the preceding record. The following are examples of the BACKSPACE statement:

```
BACKSPACE 10
BACKSPACE (11, IOSTAT=ios, ERR=100)
BACKSPACE (12, ERR=100)
BACKSPACE (13, IOSTAT=ios)
```

The ENDFILE statement writes an endfile record as the next record of the file. The following are examples of the ENDFILE statement:

```
ENDFILE 10
ENDFILE (11, IOSTAT=ios, ERR=100)
ENDFILE (12, ERR=100)
ENDFILE (13, IOSTAT=ios)
```

The REWIND statement positions the file at its initial point. The following are examples of the REWIND statement:

```
REWIND 10
REWIND (11, IOSTAT=ios, ERR=100)
REWIND (12, ERR=100)
REWIND (13, IOSTAT=ios)
REWIND (14)
```

Multithreading and Standard Fortran I/O

Multithreading is the concurrent use of multiple threads of control which operate within the same address space. Multithreading is available through DOACROSS compiler directives and through the Pthreads interface.

Standard Fortran I/O is thread-safe. The runtime I/O library performs all the needed locking to permit multiple threads to concurrently execute Fortran I/O statements. The result is proper execution of all Fortran I/O statements and the sequential execution of I/O statements issued across multiple threads to files opened for sequential access.

Chapter 3

Fortran I/O Extensions

This chapter describes additional I/O routines and statements that are available. These additional routines, known as *Fortran extensions*, perform unformatted I/O.

For details about the routines discussed in this chapter, see the individual man pages for each routine. In addition, see the reference manuals for your compiler system.

BUFFER IN/BUFFER OUT Routines

BUFFER IN and BUFFER OUT statements initiate a data transfer between the specified file or unit at the current record and the specified area of program memory. To allow maximum asynchronous performance, all BUFFER IN and BUFFER OUT operations should begin and end on a sector boundary.

The BUFFER IN and BUFFER OUT statements can perform sequential asynchronous unformatted I/O if the files are assigned as unbuffered files. You must declare the BUFFER IN and BUFFER OUT files as unbuffered by using one of the following assign(1) commands.

```
assign -s u ...
assign -F system ...
```

If the files are not declared as unbuffered, the BUFFER IN and BUFFER OUT statements may execute synchronously.

For tapes, BUFFER IN and BUFFER OUT operate synchronously; when you execute a BUFFER statement, the data is placed in the buffer before you execute the next statement in the program. Therefore, for tapes, BUFFER IN has no advantage over a read statement or a CALL READ statement; however, the library code is doing asynchronous read-aheads to fill its own buffer.

The F77 format is the default file structure.

The BUFFER IN and BUFFER OUT statements decrease the overhead associated with transferring data through library and system buffers. These statements also offer the advantages of asynchronous I/O. I/O operations for several files can execute concurrently and can also execute concurrently with CPU instructions. This can decrease overall wall-clock time.

In order for this to occur, the program must ensure that the requested asynchronous data movement was completed before accessing the data. The program must also be able to do a significant amount of CPU-intensive work or other I/O during asynchronous I/O to increase the program speed.

Buffer I/O processing waits until any previous buffer I/O operation on the file completes before beginning another buffer I/O operation.

Use the `UNIT`(3f) and `LENGTH`(3f) functions with `BUFFER IN` and `BUFFER OUT` statements to delay further program execution until the buffer I/O statement completes.

For details about the routines discussed in this section, see the individual man pages for each routine.

The `UNIT` Intrinsic

The `UNIT` intrinsic routine waits for the completion of the `BUFFER IN` or `BUFFER OUT` statement. A program that uses asynchronous `BUFFER IN` and `BUFFER OUT` must ensure that the data movement completes before trying to access the data. The `UNIT` routine can be called when the program wants to delay further program execution until the data transfer is complete. When the buffer I/O operation is complete, `UNIT` returns a status indicating the outcome of the buffer I/O operation.

The following is an example of the `UNIT` routine:

```
STATUS=UNIT(90)
```

The `LENGTH` Intrinsic

The `LENGTH` intrinsic routine returns the length of transfer for a `BUFFER IN` or a `BUFFER OUT` statement. If the `LENGTH` routine is called during a `BUFFER IN` or `BUFFER OUT` operation, the execution sequence is delayed until the transfer is complete. `LENGTH` then returns the number of words successfully transferred. A 0 is returned for an end-of-file (EOF).

The following is an example of the `LENGTH` routine:

```
LENG=LENGTH(90)
```

Named Pipe Support

Named pipes or *UNIX FIFO special files* are created with the `mknod`(2) system call; these special files allow any two processes to exchange information. The system call creates an inode for the named pipe and establishes it as a read/write named pipe. It can then be used by standard Fortran I/O or C I/O. Piped I/O is faster than normal I/O; it requires less memory than memory-resident files.

Named Pipes

After a named pipe is created, Fortran programs can access that pipe almost as if it were a typical file; the differences between process communication using named pipes and process communication using normal files is discussed in the following list. The examples show how a Fortran program can use standard Fortran I/O on pipes.

- A named pipe must be created before a Fortran program opens it. The following is the syntax for the command to create a named pipe called `fort.13`:

  ```
  /etc/mknod fort.13 p
  ```

  A named pipe can be created from within a Fortran program by using `ISHELL`(3f) or by using the C language library interface to the `mknod`(2) system call; either of the following examples creates a named pipe:

  ```
  CALL ISHELL('/etc/mknod fort.13 p')
  ```

  ```
  I = MKNOD ('fort.13',010600B,0)
  ```

- Fortran programs can communicate using two named pipes: one to read and one to write. A Fortran program must either read from or write to any named pipe, but it cannot do both at the same time. This is a Fortran restriction on pipes, not a system restriction. It occurs because Fortran does not allow read and write access at the same time.

- I/O transfers through named pipes use memory for buffering. A separate buffer is created for each named pipe that is created. The `PIPE_BUF` parameter defines the kernel buffer size in the `/sys/param.h` parameter file. The default value of `PIPE_BUF` is 8 blocks (`8 * 512` words), but the full size may not be needed or used. I/O to named pipes does not transfer to or from a disk. However, if I/O transfers fill the buffer, the writing process waits for the receiving process to read the data before refilling the buffer. If the size of the PIPE_BUF parameter is increased, I/O performance may decrease; there may be more I/O buffer contention. If memory has already been allocated for buffers, more space will not be allocated.

- Binary data transferred between two processes through a named pipe must use the correct file structure. The undefined file structure (specified by `assign -s u`) should be specified for a pipe by the sending process. The unblocked structure (specified by `assign -s unblocked`) should be specified for a pipe by the receiving process.

  The file structure for the pipe of the sending (write) process should be set to `undefined` (`assign -s u`), which issues a system call for each write. You can also select a file specification of `system` (`assign -F system`) for the sending process.

  The file structure of the receiving or read process can be set to either the undefined or the unblocked file structure. However, if the sending process writes a request that is larger than MAXPIPE, it is essential for the receiving process to read the data from a pipe set to the unblocked file structure. A read of a transfer larger than MAXPIPE on an undefined file structure yields only MAXPIPE amount of data. The receiving process would not wait to see whether the sending process is refilling the buffer. The pipe may be less than MAXPIPE.

  For example, the following `assign` commands specify that the file structure of the named pipe (unit 13, file name `pipe`) for the sending process should be `undefined (-s u)`. The named pipe (unit 15, file name `pipe`) is type `unblocked (-s unblocked)` for the read process.

  ```
  assign -s u -a pipe u:13
  assign -s unblocked -a pipe u:15
  ```

- A read from a pipe that is closed by the sender causes a detection of end-of-file (EOF).

  To detect EOF on a named pipe, the pipe must be opened as read-only by the receiving process. Use the ACTION=READ specifier on the OPEN statement to open a file as read-only.

Piped I/O Example without End-of-file Detection

In this example, two Fortran programs communicate without end-of-file (EOF) detection. In the example, program `writerd` generates an array that contains the elements 1 to 3 and writes the array to named pipe `pipe1`. Program `readwt` reads the three elements from named pipe `pipe1`, prints out the values, adds `1` to each value, and writes the new elements to named pipe `pipe2`. Program `writerd` reads the new values from named pipe `pipe2` and prints them. The `-a` option of the `assign`(1) command allows the two processes to access the same file with different `assign` characteristics.

Example 4-1 No EOF detection: `writerd`

```
        program writerd
        parameter(n=3)
        dimension ia(n)
        do 10 i=1,n
           ia(i)=i
10      continue
        write (10) ia
        read (11) ia
        do 20 i=1,n
           print*,'ia(',i,') is ',ia(i),' in writerd'
20      continue
        end
```

Example 4-2 No EOF detection: `readwt`

```
        program readwt
        parameter(n=3)
        dimension ia(n)
        read (15) ia
        do 10 i=1,n
           print*,'ia(',i,') is ',ia(i),' in readwt'
           ia(i)=ia(i)+110      continue
        write (16) ia
        end
```

The following commands execute the programs:

```
f90 -o readwt readwt.f
f90 -o writerd writerd.f
/etc/mknod pipe1 p
/etc/mknod pipe2 p
assign -s u -a pipe1 u:10
assign -s unblocked -a pipe2 u:11
assign -s unblocked -a pipe1 u:15
assign -s u -a pipe2 u:16
readwt &
writerd
```

The following is the output of the two programs:

```
ia(1) is 1 in readwt
ia(2) is 2 in readwt
ia(3) is 3 in readwt
ia(1) is 2 in writerd
ia(2) is 3 in writerd
ia(3) is 4 in writerd
```

Detecting End-of-file on a Named Pipe

The following conditions must be met to detect end-of-file on a read from a named pipe within a Fortran program: the program that sends data must open the pipe in a specific way, and the program that receives the data must open the pipe as read-only.

The program that sends or writes the data must open the named pipe as read and write or write-only. This is the default because the `/etc/mknod` command creates a named pipe with read and write permission.

The program that receives or reads the data must open the pipe as read-only. A read from a named pipe that is opened as read and write waits indefinitely for the data. Use the `ACTION=READ` specifier on the `OPEN` statement to open a file as read-only.

Piped I/O Example with End-of-file Detection

This example uses named pipes for communication between two Fortran programs with end-of-file detection. The programs in this example are similar to the programs used in the preceding section. This example shows that program `readwt` can detect the EOF.

Program `writerd` generates array `ia` and writes the data to the named pipe `pipe1`. Program `readwt` reads the data from the named pipe `pipe1`, prints the values, adds one to each value, and writes the new elements to named pipe `pipe2`. Program `writerd` reads the new values from `pipe2` and prints them. Finally, program `writerd` closes `pipe1` and causes program `readwt` to detect the EOF.

The following commands execute these programs:

```
f90 -o readwt readwt.f
f90 -o writerd writerd.f
assign -s u -a pipe1 u:10
assign -s unblocked -a pipe2 u:11
assign -s unblocked -a pipe1 u:15
assign -s u -a pipe2 u:16
/etc/mknod pipe1 p
/etc/mknod pipe2 p
readwt &
writerd
```

Example 4-3 EOF detection: `writerd`

```
      program writerd
      parameter(n=3)
      dimension ia(n)
      do 10 i=1,n
        ia(i)=i
10    continue
      write (10) ia
      read (11) ia
      do 20 i=1,n
        print*,'ia(',i,') is',ia(i),' in writerd'
20    continue
      close (10)
      end
```

Example 4-4 EOF detection: `readwt`

```
      program readwt
      parameter(n=3)
      dimension ia(n)
C     open the pipe as read-only
      open(15,form='unformatted', action='read')
      read (15,end = 101) ia
      do 10 i=1,n
        print*,'ia(',i,') is ',ia(i),' in readwt'
        ia(i)=ia(i)+1
 10   continue
      write (16) ia
      read (15,end = 101) ia
      goto 102
```

```
101   print *,’End of file detected’
102   continue
      end
```

The output of the two programs is as follows:

```
ia(1) is 1 in readwt
ia(2) is 2 in readwt
ia(3) is 3 in readwt
ia(1) is 2 in writerd
ia(2) is 3 in writerd
ia(3) is 4 in writerd
End of file detected
```

Chapter 5

System and C I/O

This chapter describes systems calls used by the I/O library to perform asynchronous or synchronous I/O. This chapter also describes Fortran callable entry points to several C library routines.

System I/O

The I/O library and programs use the system calls described in this chapter to perform synchronous and asynchronous I/O, to queue a list of distinct I/O requests, and to perform unbuffered I/O without system buffering.

Synchronous I/O

With synchronous I/O, an executing program relinquishes control during the I/O operation until the operation is complete. An operation is not complete until all data is moved.

The `read`(2) and `write`(2) system calls perform synchronous reads and writes. The `READ`(3f) and `WRITE`(3f) functions provide a Fortran interface to the `read` and `write` system calls. The `read` system call reads a specified number of bytes from a file into a specified buffer. The `write` system call writes from a buffer to a file.

Asynchronous I/O

Asynchronous I/O lets the program use the time that an I/O operation is in progress to perform some other operations that do not involve the data in the I/O operation. In asynchronous I/O operations, control is returned to the calling program after the I/O is initiated. The program may perform calculations unrelated to the previous I/O request or it may issue another unrelated I/O request while waiting for the first I/O request to complete.

The asynchronous I/O routines provide functions that let a program wait for a particular I/O request to complete. The asynchronous form of `BUFFER IN` and `BUFFER OUT` statements used with `UNIT` and `LENGTH` routines provide this type of I/O.

Unbuffered I/O

The `open`(2) system call opens a file for reading or writing. If the I/O request is well-formed and the `O_RAW` flag is set, the `read`(3f) or `write`(3f) system call reads or writes whole blocks of data directly into user space, bypassing system cache.

C I/O from Fortran

The C library provides a set of routines that constitute a user-level I/O buffering scheme to be used by C programmers.

The `getc`(3c) and `putc`(3c) inline macros process characters. The `getchar` and `putchar` macros, and the higher-level routines `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `getw`, `printf`, `puts`, `putw`, and `scanf` all use or act as if they use `getc` and `putc`. They can be intermixed.

A file with this associated buffering is called a *streams* and is associated with a pointer to a defined type `FILE`. The `fopen`(3c) routine creates descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Three open streams with constant pointers are usually declared in the `<stdio.h>` header file and are associated with `stdin`, `stdout`, and `stderr`.

Three types of buffering are available with functions that use the `FILE` type: unbuffered, fully buffered, and line buffered, as described in the following list:

- If the stream is unbuffered, no library buffer is used.
- For a fully buffered stream, data is written from the library buffer when it is filled, and read into the library buffer when it is empty.
- If the stream is line buffered, the buffer is flushed when a new line character is written, the buffer is full, or when input is requested.

The `setbuf` and `setvbuf` functions let you change the type and size of the buffers. By default, output to a terminal is line buffered, output to `stderr` is unbuffered, and all other I/O is fully buffered. See the `setbuf`(3c) man page for details.

Mixing the use of C I/O functions with Fortran I/O on the same file may have unexpected results. If you want to do this, ensure that the Fortran file structure chosen does not introduce unexpected control words and that library buffers are flushed properly before switching between types of I/O.

The following example illustrates the use of some C routines. The `assign` environment does not affect these routines.

Example 5-1 C I/O from Fortran

```
       PROGRAM STDIOEX
       INTEGER FOPEN, FCLOSE, FWRITE, FSEEK
       INTEGER FREAD, STRM
       CHARACTER*25 BUFWR, BUFRD
       PARAMETER(NCHAR=25)
C      Open the file /tmp/mydir/myfile for update
       STRM = FOPEN('/tmp/mydir/myfile','r+')
       IF (STRM.EQ.0) THEN
          STOP 'ERROR OPENING THE FILE'
       ENDIF
C      Write
       I = FWRITE(BUFWR, 1, NCHAR, STRM)
       IF (I.NE.NCHAR*1)THEN
          STOP 'ERROR WRITING FILE'
       ENDIF
C      Rewind and read the data
       I = FSEEK(STRM, 0, 0)
       IF (I.NE.0)THEN
          STOP 'ERROR REWINDING FILE'
       ENDIF
       I = FREAD(BUFRD, 1, NCHAR, STRM)
       IF (I.NE.NCHAR*1)THEN
          STOP 'ERROR READING FILE'
       ENDIF
C      Close the file
       I = FCLOSE(STRM)
       IF (I.NE.0) THEN
          STOP 'ERROR CLOSING THE FILE'
       ENDIF
       END
```

Chapter 6

The `assign` Environment

Fortran programs require the ability to alter many details of a Fortran file connection. You may need to specify device residency, an alternative file name, a file space allocation scheme, file structure, or data conversion properties of a connected file.

This chapter describes the `assign`(1) command and the `ASSIGN`(3f) library routine, which are used for these purposes. The `ffassign` command provides an interface to assign processing from C. See the `ffassign` man page for details about its use.

`assign` Basics

The `assign`(1) command passes information to Fortran `OPEN` statements and to the `ffopen`(3c), `AQOPEN`(3f), `WOPEN`(3f), `OPENDR`(3f), and `OPENMS`(3f) routines.

This information is called the *assign environment*; it consists of the following elements:

- A list of unit numbers
- File names
- File name patterns that have attributes associated with them

Any file name, file name pattern, or unit number to which assign options are attached is called an *assign_object*. When the unit or file is opened from Fortran, the options are used to set up the properties of the connection.

Open Processing

The I/O library routines apply options to a file connection for all related *assign_object*s.

If the *assign_object* is a unit, the application of options to the unit occurs whenever that unit becomes connected.

If the *assign_object* is a file name or pattern, the application of options to the file connection occurs whenever a matching file name is opened from a Fortran program.

When any of the previously listed library I/O routines open a file, they use `assign` options for any *assign_object*s which apply to this open request. Any of the following *assign_object*s or categories might apply to a given open request:

- `g:all` options apply to any open request.
- `g:su`, `g:sf`, `g:du`, , and `g:ff` each apply to types of open requests (for example, sequential unformatted, sequential formatted, and so on).
- `u:`*unit_number* applies whenever unit *unit_number* is opened.
- `p:`*pattern* applies whenever a file whose name matches *pattern* is opened. The `assign` environment can contain only one `p:` *assign_object* which matches the current open file. The exception is that the `p:%`*pattern* (which uses the `%` wildcard character) is silently ignored if a more specific *pattern* also matches the current filename being opened.
- `f:`*filename* applies whenever a file with the name *filename* is opened.

Options from the assign objects in these categories are collected to create the complete set of options used for any particular open. The options are collected in the listed order, with options collected later in the list of `assign` objects overriding those collected earlier.

The `assign` Command

The following is the syntax for the `assign` command:

```
assign [-a actualfile] [-b bs] [-f fortstd] [-s ft] [-t] [-y setting] [-B setting]
   [-C charcon] [-D fildes] [-F spec[,specs]] [-I] [-N numcon] [-O] [-R]
   [-S setting] [-T setting] [-U setting] [-V] [-W setting] [-Y setting] [-Z setting]
   assign_object
```

The following two specifications cannot be used with any other options:

```
assign -R [assign_object]
```

```
assign -V [assign_object]
```

The following is a summary of the `assign` command options. For details, see the `assign`(1) and `INTRO_FFIO`(3f) man pages.

`-I` Specifies an incremental assign. All attributes are added to the attributes already assigned to the current *assign_object*. This option and the `-O` option are mutually exclusive.

`-O`	Specifies a replacement assign. This is the default control option. All currently existing `assign` attributes for the current *assign_object* are replaced. This option and the `-I` option are mutually exclusive.
`-R`	Removes all `assign` attributes for *assign_object*. If *assign_object* is not specified, all currently assigned attributes for all *assign_object*s are removed.
`-V`	Views attributes for *assign_object*. If *assign_object* is not specified, all currently assigned attributes for all *assign_object*s are printed.

The following are the `assign` command attribute options:

`-a` *actualfile*	The `FILE=` specifier or the actual file name.
`-b` *bs*	Library buffer size in 4096–byte blocks.
`-f` *fortstd*	Fortran standard. Specify `77` to be compatible with the FORTRAN 77 standard. Specify `90` to be compatible with the Fortran 90 standard. Specify `irixf77` to be compatible with SGI's FORTRAN 77 compiling system which runs on IRIX systems. Specify `irixf90` to be compatible with the MIPSpro 7 Fortran 90 compiler.
`-s` *ft*	File type. Enter `text`, `cos`, `blocked`, `unblocked`, `u`, `sbin`, or `bin` for *ft*.
`-t`	Temporary file.
`-y` *setting*	Suppresses repeat counts in list-directed output. *setting* can be either `on` or `off`. The default setting is `off`.
`-B` *setting*	Activates or suppresses the passing of the `O_DIRECT` flag to the `open`(2) system call. Enter either `on` or `off` for *setting*.
`-C` *charcon*	Character set conversion information. Enter `ascii` for *charcon*. If you specify the `-C` option, you must also specify the `-F` option.

`-D` *fildes*	Specifies a connection to a standard file. Enter `stdin`, `stdout`, or `stderr` for *fildes*.
`-F` *spec* `[,`*specs*`]`	Flexible file I/O (FFIO) specification. See the `assign`(1) man page for details about allowed values for *spec* and for details about hardware platform support. See the `INTRO_FFIO`(3f) man page for details about specifying the FFIO layers.
`-N` *numcon*	Foreign numeric conversion specification. See the `assign`(1) man page for details about allowed values for *numcon* and for details about hardware platform support.
`-S` *setting*	Suppresses use of a comma as a separator in list-directed output. Enter either `on` or `off` for *setting*. The default setting is `off`.
`-T` *setting*	Activates or suppresses truncation after write for sequential Fortran files. Enter either `on` or `off` for *setting*.
`-U` *setting*	Produces a form of list-directed output. This is a global setting which sets the value for the `-y`, `-S`, and `-W` options. Enter either `on` or `off` for *setting*. The default setting is `off`.
`-W` *setting*	Suppresses compressed width in list-directed output. Enter either `on` or `off` for *setting*. The default setting is `off`.
`-Y` *setting*	Skips unmatched namelist groups in a namelist input record. Enter either `on` or `off` for *setting*. The default setting is `on`.
`-Z` *setting*	Recognizes –0.0 for IEEE floating point systems and writes the minus sign for edit-directed, list-directed, and namelist output. Enter either `on` or `off` for *setting*. The default setting is `off`.
assign_object	Specifies either a file name or a unit number for *assign_object*. The `assign` command associates the attributes with the file or unit specified. These

attributes are used during the processing of Fortran `OPEN` statements or during implicit file opens.

Use one of the following formats for *assign_object*:

- `f:`*file_name* (for example, `f:file1`)
- `g:`*io_type*; *io_type* can be `su`, `sf`, `du`, `df`, `ff`, or `aq` (for example, `g:ff`)
- `p:`*pattern* (for example, `p:file%`)
- `u:`*unit_number* (for example, `u:9`)
- *file_name* (for example, `myfile`)

When the `p`: *pattern* form is used, the `%` and `_` wildcard characters can be used. The `%` matches any string of 0 or more characters. The `_` matches any single character. The `%` performs like the `*` when doing file name matching in shells. However, the `%` character also matches strings of characters containing the `/` character.

Related Library Routines

The `ASSIGN`(3f), `ASNUNIT`(3f), `ASNFILE`(3f), and `ASNRM`(3f) routines can be called from a Fortran program to access and update the assign environment. The `ASSIGN` routine provides an easy interface to `ASSIGN` processing from a Fortran program. The `ASNUNIT` and `ASNFILE` routines assign attributes to units and files, respectively. The `ASNRM` routine removes all entries currently in the `assign` environment.

The calling sequences for the `assign` library routines are as follows:

`CALL ASSIGN (`*cmd*`,`*ier*`)`

`CALL ASNUNIT (`*iunit*`,`*astring*`,`*ier*`)`

`CALL ASNFILE (`*fname*`,`*astring*`,`*ier*`)`

`CALL ASNRM (`*ier*`)`

cmd	Fortran character variable that contains a complete `assign` command in the format that is also acceptable to the `ISHELL`(3f) routine.
ier	Integer variable that is assigned the exit status on return from the library interface routine.

iunit	Integer variable or constant that contains the unit number to which attributes are assigned.
astring	Fortran character variable that contains any attribute options and option values from the `assign` command. Control options `-I`, `-O`, and `-R` can also be passed.
fname	Character variable or constant that contains the file name to which attributes are assigned.

A status of 0 indicates normal return and a status of greater than 0 indicates a specific error status. Use the `explain` command to determine the meaning of the error status. For more information about the `explain` command, see the `explain`(1) man page.

The following calls are equivalent to the `assign -s u f:file` command:

```
CALL ASSIGN('assign -s u f:file',ier)
CALL ASNFILE('file','-s u',IER)
```

The following call is equivalent to executing the `assign -I -n 2 u:99` command:

```
IUN = 99
CALL ASNUNIT(IUN,'-I -n 2',IER)
```

The following call is equivalent to executing the `assign -R` command:

```
CALL ASNRM(IER)
```

`assign` and Fortran I/O

Assign processing lets you tune file connections. The following sections describe several areas of `assign` command usage and provide examples of each use.

Alternative File Names

The `-a` option specifies the actual file name to which a connection is made. This option allows files to be created in alternative directories without changing the `FILE=` specifier on an `OPEN` statement.

For example, consider the following `assign` command issued to open unit 1:

```
assign -a /tmp/mydir/tmpfile u:1
```

The program then opens unit 1 with any of the following statements:

```
WRITE(1) variable          ! implicit open
OPEN(1)                    ! unnamed open
OPEN(1,FORM='FORMATTED')   ! unnamed open
```

Unit 1 is connected to file `/tmp/mydir/tmpfile`. Without the `-a` attribute, unit 1 would be connected to file `fort.1`.

When the `-a` attribute is associated with a file, any Fortran open that is set to connect to the file causes a connection to the actual file name. An `assign` command of the following form causes a connection to file `$TMPDIR/joe`:

```
assign -a $TMPDIR/joe ftfile
```

This is true when any of the following statements are executed in a program:

```
OPEN(IUN,FILE='ftfile')
CALL AQOPEN(AQP,AQPSIZE,'ftfile',ISTAT)
CALL OPENMS('ftfile',INDARR,LEN,IT)
CALL OPENDR('ftfile',INDARR,LEN,IT)
CALL WOPEN('ftfile',BLOCKS,ISTATS)
WRITE('ftfile') ARRAY
```

If the following `assign` command is issued and is in effect, any Fortran `INQUIRE` statement whose `FILE=` specification is `foo` refers to the file named `actual` instead of the file named `foo` for purposes of the `EXISTS=`, `OPENED=`, or `UNIT=` specifiers:

```
assign -a actual f:foo
```

If the following `assign` command is issued and is in effect, the `-a` attribute does not affect `INQUIRE` statements with a `UNIT=` specifier:

```
assign -a actual ftfile
```

When the following `OPEN` statement is executed, `INQUIRE(UNIT=`*n*`,NAME=`*fname*`)` returns a value of `ftfile` in *fname*, as if no `assign` had occurred:

```
OPEN(n,file='ftfile')
```

The I/O library routines use only the actual file (`-a`) attributes from the `assign` environment when processing an `INQUIRE` statement. During an `INQUIRE` statement that contains a `FILE=` specifier, the I/O library searches the `assign` environment for a reference to the file name that the `FILE=` specifier supplies. If an *assign-by-filename* exists for the file name, the I/O library determines whether an actual name from the `-a` option is associated with the file name. If the *assign-by-filename* supplied an actual name, the I/O library uses the name to return values for the `EXIST=`, `OPENED=`, and `UNIT=` specifiers; otherwise, it uses the file name. The name returned for the `NAME=` specifier is the file name supplied in the `FILE=` specifier. The actual file name is not returned.

File Structure Selection

Fortran I/O uses the `text` file structure, unblocked file structure, `pure` file structure, F77 file structure, and COS blocked structure. By default, a file structure is selected for a unit based on the type of Fortran I/O selected at open time. If an alternative file structure is needed, the user can select a file structure by using the `-s` and `-F` options on the `assign` command.

No *assign_object* can have both `-s` and `-F` attributes associated with it. Some file structures are available as `-F` attributes but are not available as `-s` attributes. The `-F` option is more flexible than the `-s` option; it allows nested file structures and buffer size specifications for some attribute values. The following list summarizes how to select the different file structures with different options to the `assign` command:

Structure	`assign` **command**

COS blocked	`assign -F cos`
	`assign -s cos`

`text`	`assign -F text`
	`assign -s text`

unblocked	`assign -F system`
	`assign -s unblocked`
	`assign -s u`

F77 blocked	`assign -F f77`

For more information about file structures, see Chapter 7, "File Structures ", page 43.

The following are examples of file structure selection:

- To select unblocked file structure for a sequential unformatted file:

  ```
  IUN = 1
  CALL ASNUNIT(IUN,'-s unblocked',IER)
  ```

```
OPEN(IUN,FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
```

- You can use the `assign -s u` command to specify the unblocked file structure for a sequential unformatted file. When this option is selected, the I/O is unbuffered. Each Fortran `READ` or `WRITE` statement results in a `read`(2) or `write`(2) system call such as the following:

  ```
  CALL ASNFILE('fort.1','-s u',IER)
  OPEN(1,FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
  ```

- Use the following command to assign unit 10 a COS blocked structure:

  ```
  assign -F cos u:10
  ```

Buffer Size Specification

The size of the buffer used for a Fortran file can have a substantial effect on I/O performance. A larger buffer size usually decreases the system time needed to process sequential files. However, large buffers increase a program's memory usage; therefore, optimizing the buffer size for each file accessed in a program on a case-by-case basis can help increase I/O performance and can minimize memory usage.

The `-b` option on the `assign` command specifies a buffer size, in blocks, for the unit. The `-b` option can be used with the `-s` option, but it cannot be used with the `-F` option. Use the `-F` option to provide I/O path specifications that include buffer sizes; the `-b`, and `-u` options do not apply when `-F` is specified.

For more information about the selection of buffer sizes, see Chapter 8, "Buffering", page 51, and the `assign`(1) man page.

The following are some examples of buffer size specification using the `assign -b` and `assign -F` options:

- If unit 1 is a large sequential file for which many Fortran `READ` or `WRITE` statements are issued, you can increase the buffer size to a large value, using the following `assign` command:

  ```
  assign -b 336 u:1
  ```

- If file `foo` is a small file or is accessed infrequently, minimize the buffer size using the following `assign` command:

  ```
  assign -b 1 f:foo
  ```

Foreign File Format Specification

The Fortran I/O library can read and write files with record blocking and data formats native to operating systems from other vendors. The `assign -F` command specifies a foreign record blocking; the `assign -C` command specifies the type of character conversion; the `-N` option specifies the type of numeric data conversion. When `-N` or `-C` is specified, the data is converted automatically during the processing of Fortran `READ` and `WRITE` statements. For example, assume that a record in file `fgnfile` contains the following character and integer data:

```
character*4 ch
integer int
open(iun,FILE='fgnfile',FORM='UNFORMATTED')
read(iun) ch, int
```

Use the following `assign` command to specify foreign record blocking and foreign data formats for character and integer data:

```
assign -F ibm.vbs -N ibm -C ebcdic fgnfile
```

Direct-access I/O Tuning

Fortran unformatted direct-access I/O supports number tuning and memory cache page size (buffer) tuning; it also supports specification of the prevailing direction of file access. The `assign -b` command specifies the size of each buffer in 4096–byte blocks.

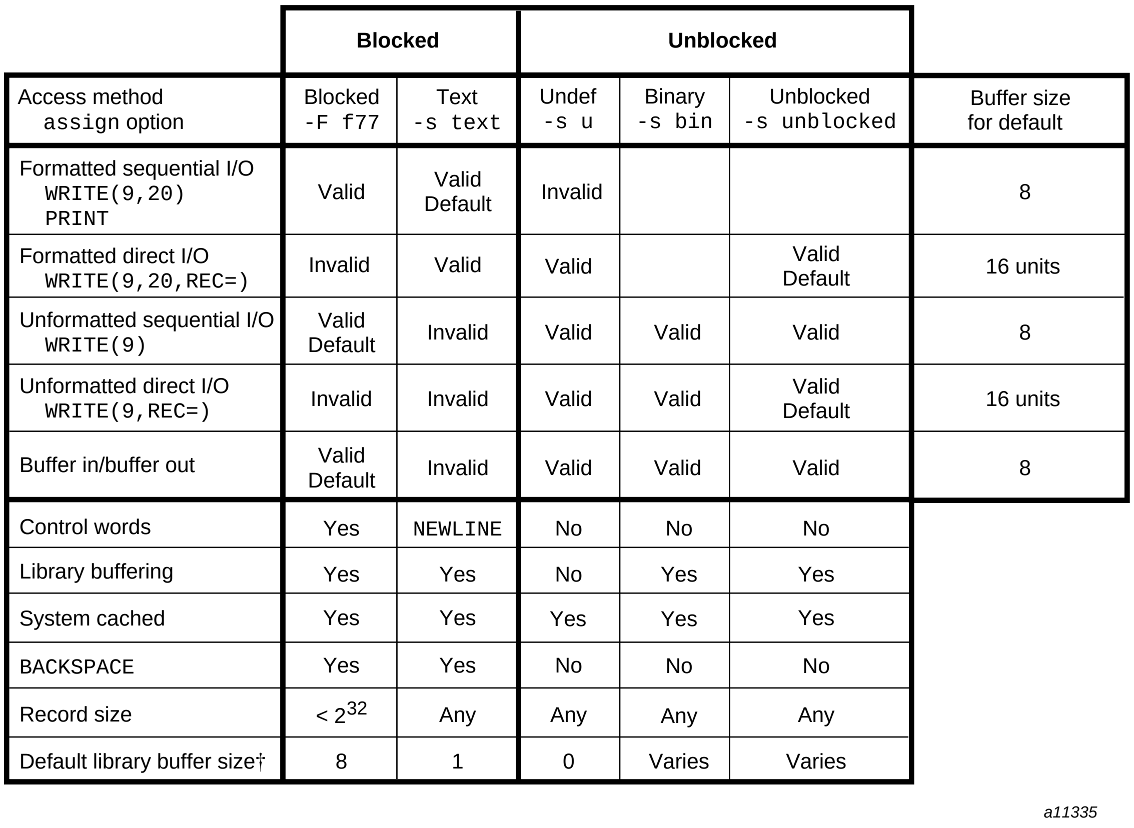
Fortran File Truncation

The `assign -T` option activates or suppresses truncation after the writing of a sequential Fortran file. The `-T on` option specifies truncation; this behavior is consistent with the Fortran standard and is the default setting for most `assign -s` *fs* specifications.

The `assign`(1) man page lists the default setting of the `-T` option for each `-s fs` specification. It also indicates if suppression or truncation is allowed for each of these specifications.

FFIO layers that are specified by using the `-F` option vary in their support for suppression of truncation with `-T off`.

The following figure summarizes the available access methods and the default buffer sizes. Figures are given in units of 4096 bytes.



	Blocked		Unblocked			
Access method `assign` option	Blocked `-F f77`	Text `-s text`	Undef `-s u`	Binary `-s bin`	Unblocked `-s unblocked`	Buffer size for default
Formatted sequential I/O `WRITE(9,20)` `PRINT`	Valid	Valid Default	Invalid			8
Formatted direct I/O `WRITE(9,20,REC=)`	Invalid	Valid	Valid		Valid Default	16 units
Unformatted sequential I/O `WRITE(9)`	Valid Default	Invalid	Valid	Valid	Valid	8
Unformatted direct I/O `WRITE(9,REC=)`	Invalid	Invalid	Valid	Valid	Valid Default	16 units
Buffer in/buffer out	Valid Default	Invalid	Valid	Valid	Valid	8
Control words	Yes	`NEWLINE`	No	No	No	
Library buffering	Yes	Yes	No	Yes	Yes	
System cached	Yes	Yes	Yes	Yes	Yes	
`BACKSPACE`	Yes	Yes	No	No	No	
Record size	$< 2^{32}$	Any	Any	Any	Any	
Default library buffer size†	8	1	0	Varies	Varies	

Figure 6-1 Access methods and default buffer size (IRIX systems)

The `assign` Environment File

To use the `assign` command, you must set the `FILENV` environment variable. `FILENV` can contain the pathname of a file which will be used to store `assign` information or it can specify that the information should be stored in the process environment.

Local assign

The assign environment information is usually stored in the assign environment file. Programs that do not require the use of the global assign environment file can activate local assign mode. If you select local assign mode, the assign environment will be stored in memory. Thus, other processes could not adversely affect the assign environment used by the program.

The ASNCTL(3f) routine selects local assign mode when it is called by using one of the following command lines:

```
CALL ASNCTL('LOCAL',1,IER)
CALL ASNCTL('NEWLOCAL',1,IER)
```

Example 6-1 local assign mode

In the following example, a Fortran program activates local assign mode and then specifies an unblocked data file structure for a unit before opening it. The -I option is passed to ASNUNIT to ensure that any assign attributes continue to have an effect at the time of file connection.

```
C    Switch to local assign environment
     CALL ASNCTL('LOCAL',1,IER)
     IUN = 11
C    Assign the unblocked file structure
     CALL ASNUNIT(IUN,'-I -s unblocked',IER)
C    Open unit 11
     OPEN(IUN,FORM='UNFORMATTED')
```

If a program contains all necessary assign statements as calls to ASSIGN, ASNUNIT, and ASNFILE, or if a program requires total shielding from any assign commands, use the second form of a call to ASNCTL, as follows:

```
C    New (empty) local assign environment
     CALL ASNCTL('NEWLOCAL',1,IER)
     IUN = 11
C    Assign a large buffer size
     CALL ASNUNIT(IUN,'-b 336',IER)
C    Open unit 11
     OPEN(IUN,FORM='UNFORMATTED')
```

File Structures

A file structure defines the way that records are delimited and how the end-of-file is represented.

The unblocked, pure, text, and F77 file structures can be used.

The I/O library provides four different forms of file processing to indicate an unblocked file structure by using the `assign -s` *ft* command: unblocked (`unblocked`), standard binary (`sbin`), binary (`bin`), and undefined (`u`). These alternative forms provide different types of I/O packages used to access the records of the file, different types of file truncation and data alignment, and different endfile record recognitions in a file.

The full set of options allowed with the `assign -s` *ft* command are the following:

- `bin` (not recommended)
- `blocked`
- `cos`
- `sbin`
- `text`
- `u`
- `unblocked`

For more information about valid arguments to the `assign -F` command, see "File Structure Selection", page 38. Table 7-1 summarizes the Fortran access methods and options.

Table 7-1 Fortran access methods and options

Access and form	`assign -s` *ft* defaults	`assign -s` *ft* options

Unformatted sequential `BUFFER IN` / `BUFFER OUT`	`blocked`	`bin`
		`sbin`
		`u`
		`unblocked`

Unformatted direct	`unblocked`	`bin`
		`sbin`
		`u`
		`unblocked`

Formatted sequential	`text`	`blocked`
		`cos`
		`sbin/text`

Formatted direct on IRIX systems	`unblocked`	`u` `unblocked`

You cannot specify the default for unformatted sequential access with `assign -s`. You must use `assign -F f77`.

Unblocked File Structure

A file with an unblocked file structure contains undelimited records. Because it does not contain any record control words, it does not have record boundaries. The unblocked file structure can be specified for a file that is opened with either unformatted sequential access or unformatted direct access. It is the default file structure for a file opened as an unformatted direct-access file.

If a file with unblocked file structure must be repositioned, a `BACKSPACE` statement should not be used. You cannot reposition the file to a previous record when record boundaries do not exist.

`BUFFER IN` and `BUFFER OUT` statements can specify a file that is an unbuffered and unblocked file structure. If the file is specified with `assign -s u`, `BUFFER IN` and `BUFFER OUT` statements can perform asynchronous unformatted I/O.

You can specify the unblocked data file structure by using the `assign`(1) command in several ways. All methods result in a similar file structure but with different library buffering styles, use of truncation on a file, alignment of data, and recognition of an endfile record in the file. The following unblocked data file structure specifications are available:

Specification	Structure
`assign -s unblocked`	Library-buffered
`assign -F system`	No library buffering
`assign -s u`	No library buffering
`assign -s sbin`	Standard-I/O-compatible buffering; for example, both library and system buffering

The type of file processing for an unblocked data file structure depends on the `assign -s` *ft* option declared or assumed for a Fortran file.

`assign -s unblocked` File Processing

An I/O request for a file specified using the `assign -s unblocked` command does not need to be a multiple of a specific number of bytes. Such a file is truncated after the last record is written to the file. Padding occurs for files specified with the `assign -s bin` command and the `assign -s unblocked` command. Padding usually occurs when noncharacter variables follow character variables in an unformatted direct-access file.

No padding is done in an unformatted sequential access file. An unformatted direct-access file contains records that are the same length. The endfile record is recognized in sequential-access files.

`assign -s sbin` File Processing (Not Recommended)

You can use an `assign -s sbin` specification for a Fortran file that is opened with either unformatted direct access or unformatted sequential access. The file does not contain record delimiters. The file created for `assign -s sbin` in this instance has an unblocked data file structure and uses unblocked file processing.

The `assign -s sbin` option can be specified for a Fortran file that is declared as formatted sequential access. Because the file contains records that are delimited with the new-line character, it is not an unblocked data file structure. It is the same as a text file structure.

The `assign -s sbin` option is compatible with the standard C I/O functions. See Chapter 5, "System and C I/O ", page 27, for more details.

Note: Use of `assign -s sbin` is discouraged. Use `assign -s text` for formatted files, and `assign -s unblocked` for unformatted files.

`assign -s bin` File Processing (Not Recommended)

An I/O request for a file that is specified with `assign -s bin` does not need to be a multiple of a specific number of bytes. The I/O library uses an internal buffer for the records. If opened for sequential access, a file is not truncated after each record is written to the file.

`assign -s u` File Processing

The `assign -s u` command specifies undefined or unknown file processing. An `assign -s u` specification can be specified for a Fortran file that is declared as unformatted sequential or direct access. Because the file does not contain record delimiters, it has an unblocked data file structure. Both synchronous and asynchronous `BUFFER IN` and `BUFFER OUT` processing can be used with `u` file processing.

For best performance, a Fortran I/O request on a file assigned with the `assign -s u` command should be a multiple of 4096 bytes. I/O requests are not library buffered. They cause an immediate system call.

Fortran sequential files declared by using `assign -s u` are not truncated after the last word written. The user must execute an explicit `ENDFILE` statement on the file to get truncation.

Text File Structure

The text file structure consists of a stream of 8-bit ASCII characters. Every record in a text file is terminated by a newline character (`\n`, ASCII 012). Some utilities may omit the newline character on the last record, but the Fortran library will treat such an occurrence as a malformed record. This file structure can be specified for a file that is declared as formatted sequential access or formatted direct access. It is the default file structure for formatted sequential access files.

The `assign -s text` command specifies the library-buffered text file structure. Both library and system buffering are done for all text file structures (for more information about library buffering, see Chapter 8, "Buffering", page 51).

An I/O request for a file using `assign -s text` does not need to be a multiple of a specific number of bytes.

You cannot use `BUFFER IN` and `BUFFER OUT` statements with this structure. Use a `BACKSPACE` statement to reposition a file with this structure.

COS or Blocked File Structure

The cos or blocked file structure uses control words to mark the beginning of each 4096–byte block and to delimit each record. You can specify this file structure for a file that is declared as unformatted sequential access. Synchronous `BUFFER IN` and `BUFFER OUT` statements can create and access files with this file structure.

You can specify this file structure with one of the following `assign`(1) commands:

```
assign -s cos
assign -s blocked
assign -F cos
assign -F blocked
```

These four `assign` commands result in the same file structure.

An I/O request on a blocked file is library buffered. For more information about library buffering, see Chapter 8, "Buffering", page 51.

In a COS file structure, one or more `ENDFILE` records are allowed. `BACKSPACE` statements can be used to reposition a file with this structure.

A blocked file is a stream of words that contains control words called Block Control Word (BCW) and Record Control Words (RCW) to delimit records. Each record is terminated by an EOR (end-of-record) RCW. At the beginning of the stream, and every 512 words thereafter, (including any RCWs), a BCW is inserted. An end-of-file (EOF) control word marks a special record that is always empty. Fortran considers this empty record to be an endfile record. The end-of-data (EOD) control word is always the last control word in any blocked file. The EOD is always immediately preceded by an EOR, or an EOF and a BCW.

Each control word contains a count of the number of data words to be found between it and the next control word. In the case of the EOD, this count is 0. Because there is a BCW every 512 words, these counts never point forward more than 511 words.

A record always begins at a word boundary. If a record ends in the middle of a word, the rest of that word is zero filled; the `ubc` field of the closing RCW contains the number of unused bits in the last word.

The following is a representation of the structure of a BCW:

`m`	unused	`bdf`	unused	`bn`	`fwi`
(4)	(7)	(1)	(19)	(24)	(9)

Field	Bits	Description
`m`	0-3	Type of control word; 0 for BCW
`bdf`	11	Bad Data flag (1-bit).
`bn`	31-54	Block number (modulo 2^{24}).
`fwi`	55-63	Forward index; the number of words to next control word.

The following is a representation of the structure of an RCW:

`m`	`ubc`	`tran`	`bdf`	`srs`	unused	`pfi`	`pri`	`fwi`
(4)	(6)	(1)	(1)	(1)	(7)	(20)	(15)	(9)

Field	Bits	Description
m	0-3	Type of control word; 10_8 for EOR, 16_8 for EOF, and 17_8 for EOD.
ubc	4-9	Unused bit count; number of unused low-order bits in last word of previous record.
tran	10	Transparent record field (unused).
bdf	11	Bad data flag (unused).
srs	12	Skip remainder of sector (unused).
pfi	20-39	Previous file index; offset modulo 2^{20} to the block where the current file starts (as defined by the last EOF).
pri	40-54	Previous record index; offset modulo 2^{15} to the block where the current record starts.
fwi	55-63	Forward index; the number of words to next control word.

Chapter 8

Buffering

This chapter provides an overview of buffering and a description of file buffering as it applies to I/O.

Buffering Overview

I/O is the process of transferring data between a program and an external device. The process of optimizing I/O consists primarily of making the best possible use of the slowest part of the path between the program and the device.

The slowest part is usually the physical channel, which is often slower than the CPU or a memory-to-memory data transfer. The time spent in I/O processing overhead can reduce the amount of time that a channel can be used, thereby reducing the effective transfer rate. The biggest factor in maximizing this channel speed is often the reduction of I/O processing overhead.

A *buffer* is a temporary storage location for data while the data is being transferred. A buffer is often used for the following purposes:

- Small I/O requests can be collected into a buffer, and the overhead of making many relatively expensive system calls can be greatly reduced.

  A collection buffer of this type can be sized and handled so that the actual physical I/O requests made to the operating system match the physical characteristics of the device being used.

- Many data file structures, such as the `f77` and `cos` file structures, contain control words. During the write process, a buffer can be used as a work area where control words can be inserted into the data stream (a process called *blocking*). The blocked data is then written to the device. During the read process, the same buffer work area can be used to examine and remove these control words before passing the data on to the user (*deblocking*).

- When data access is random, the same data may be requested many times. A *cache* is a buffer that keeps old requests in the buffer in case these requests are needed again. A cache that is sufficiently large and/or efficient can avoid a large part of the physical I/O by having the data ready in a buffer. When the data is often found in the cache buffer, it is referred to as having a high *hit rate*. For example, if the entire file fits in the cache and the file is present in the cache, no more physical requests are required to perform the I/O. In this case, the hit rate is 100%.

- Running the disks and the CPU in parallel often improves performance; therefore, it is useful to keep the CPU busy while data is being moved. To do this when writing, data can be transferred to the buffer at memory-to-memory copy speed and an asynchronous I/O request can be made. The control is then immediately returned to the program, which continues to execute as if the I/O were complete (a process called *write-behind*). A similar process can be used while reading; in this process, data is read into a buffer before the actual request is issued for it. When it is needed, it is already in the buffer and can be transferred to the user at very high speed. This is another form or use of a cache.

The I/O path is divided into two parts. One part includes the user data area, the library buffer, and the system cache. The second part is referred to as the *logical device*, which includes the ultimate I/O device and all of the buffering, caching, and processing associated with that device. This includes any caching in the disk controller and the operating system.

Users can directly or indirectly control some buffers. These include most library buffers and, to some extent, system cache and `ldcache`. Some buffering, such as that performed in the IOS, or the disk controllers, is not under user control.

A *well-formed request* requires the following:

- The size of the request must be a multiple of the sector size in bytes. For most disk devices, this will be 4096 bytes.
- The data that will be transferred must be located on a word boundary.
- The file must be positioned on a sector boundary. This will be a 4096-byte sector boundary for most disks.

Types of Buffering

The following sections briefly describe unbuffered I/O, library buffering, and system cache buffering.

Unbuffered I/O

The simplest form of buffering is none at all; this unbuffered I/O is known as *raw I/O*. For sufficiently large, well-formed requests, buffering is not necessary; it can add unnecessary overhead and delay. The following `assign`(1) command specifies unbuffered I/O:

```
assign -s u  ...
```

Use the `assign` command to bypass library buffering for all well-formed requests. The data is transferred directly between the user data area and the logical device. Requests that are not well formed use system cache.

Library Buffering

The term *library buffering* refers to a buffer that the I/O library associates with a file. When a file is opened, the I/O library checks the access, form, and any attributes declared on the `assign` or `asgcmd`(1) command to determine the type of processing that should be used on the file. Buffers are usually an integral part of the processing.

If the file is assigned with one of the following options, library buffering is used:

`-s blocked`
`-F` *spec* (buffering as defined by *spec*)
`-s cos`
`-s bin`
`-s unblocked`

The `-F` option specifies flexible file I/O (FFIO), which uses library buffering if the specifications selected include a need for some buffering. In some cases, more than one set of buffers might be used in processing a file.

System Cache

The operating system or kernel uses a set of buffers in kernel memory for I/O operations. These are collectively called the *system cache*. The I/O library uses system calls to move data between the user memory space and the system buffer. The system cache ensures that the actual I/O to the logical device is well formed, and it tries to remember recent data in order to reduce physical I/O requests. In many cases, though, it is desirable to bypass the system cache and to perform I/O directly between the user's memory and the logical device.

For the `assign -s cos` and `assign -s bin` commands, a library buffer ensures that the actual system calls are well formed. This is not true for the `assign -s u` option. If you plan to bypass the system cache, all requests go through the cache except those that are well-formed.

See the explanation of the `-B` option on the `assign`(1) man page for information about bypassing system buffering on IRIX systems.

Default Buffer Sizes

The Fortran I/O library automatically selects default buffer sizes. These defaults can be overridden with the `assign` command.

The default buffer sizes are as follows (note that one block is 4096 bytes):

Sequential access, formatted	Default buffer size is 8 blocks.
Sequential access, unformatted	Default buffer size is 8 blocks.
Direct access, formatted	Default buffer size is 16 blocks.
Direct access, unformatted	Default buffer size is 16 blocks. Four buffers of this size are allocated.

Introduction to FFIO

This chapter provides an overview of the capabilities of the *flexible file input/output* (FFIO) system, sometimes called the FFIO system or *layered input/output* (I/O). The FFIO system is used to perform many I/O-related tasks. For details about each individual I/O layer, see Chapter 13, "FFIO Layer Reference ", page 89.

Layered I/O

The FFIO system is based on the concept that for all I/O a list of processing steps must be performed to transfer the user data between the user's memory and the desired I/O device. Computer manufacturers have always provided I/O options to users because I/O is often the slowest part of a computational process. In addition, it is extremely difficult to provide one I/O access method that works optimally in all situations.

The following figure depicts the typical flow of data from the user's variables to and from the I/O device.

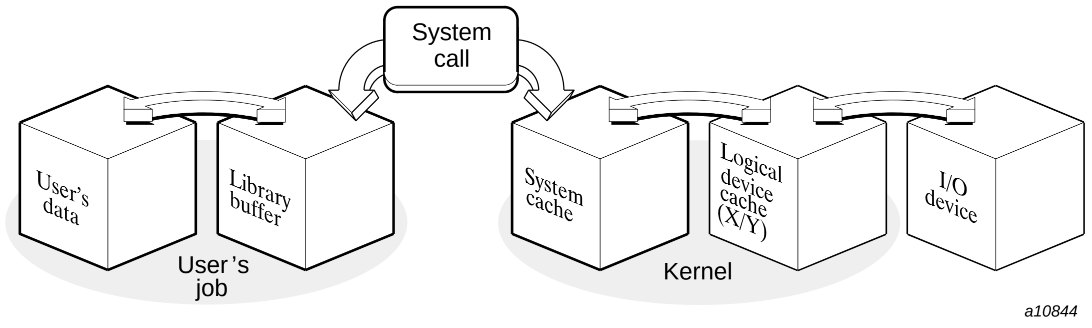


Figure 9-1 Typical data flow

It is useful to think of each of these boxes as a stopover for the data, and each transition between stopovers as a processing step.

Each transition has benefits and costs. Different applications might use the total I/O system in different ways. For example, if I/O requests are large, the library buffer is unnecessary because the buffer is used primarily to avoid making system calls for every small request. You can achieve better I/O throughput with large I/O requests by not using library buffering.

If library buffering is not used, I/O requests should be on 4096–byte block boundaries; otherwise, I/O performance will be degraded. On the other hand, if all I/O requests are very small, the library buffer is essential to avoid making a costly system call for each I/O request.

It is useful to be able to modify the I/O process to prevent intermediate steps (such as buffering of data) for existing programs without requiring that the source code be changed. The `assign`(1) command lets you modify the total user I/O path by establishing an I/O environment.

The FFIO system lets you specify each stopover in Figure 9-1, page 55. You can specify a comma-separated list of one or more processing steps by using the `assign -F` command:

```
assign -F spec1,spec2,spec3...
```

Each *spec* in the list is a processing step that requests one I/O *layer,* or logical grouping of layers. The layer specifies the operations that are performed on the data as it is passed between the user and the I/O device. A *layer* refers to the specific type of processing being done. In some cases, the name corresponds directly to the name of one layer. In other cases, however, specifying one layer invokes the routines used to pass the data through multiple layers. See the `INTRO_FFIO`(3f) man page for details about using the `-F` option to the `assign` command.

Processing steps are ordered as if the `-F` side (the left side) is the user and the system/device is the right side, as in the following example:

```
assign -F user,bufa,system
```

With this specification, a `WRITE` operation first performs the `user` operation on the data, then performs the `bufa` operation, and then sends the data to the system. In a `READ` operation, the process is performed from right to left. The data moves from the system to the user. The layers closest to the user are *higher-level layers*; those closer to the system are *lower-level layers*.

The FFIO system has an internal model of the world of data, which it maps to any given actual logical file type. Four of these concepts are basic to understanding the inner workings of the layers.

Concept	**Definition**
Data	Data is a stream of bits.
Record marks	End-of-record marks (EOR) are boundaries between logical records.
File marks	End-of-file marks (EOF) are special types of record marks that exist in some file formats.
End-of-data (EOD)	An end-of-data (EOD) is a point immediately beyond the last data bit, EOR, or EOF in the file.

All files are streams of 0 or more bits that may contain record or file marks.

Individual layers have varying rules about which of these things can appear and in which order they can appear in a file.

Fortran users can use the `assign`(1) command to specify these FFIO options. For C users, the FFIO layers are available only to programs that call the FFIO routines directly (`ffopen`(3c), `ffread`(3c), and `ffwrite`(3c)).

You can use FFIO with the following Fortran I/O forms:

- Buffer I/O
- Unformatted sequential
- Unformatted direct access
- Formatted sequential
- Namelist
- List-directed

Using Layered I/O

The specification list on the `assign -F` command comprises all of the processing steps that the I/O system performs. If `assign -F` is specified, any default processing is overridden. The FFIO system provides detailed control over I/O processing requests. However, to effectively use any FFIO option, you must understand the I/O processing details.

As a very simple example, suppose you were making large I/O requests and did not require buffering or blocking on your data. You could specify the following:

```
assign -F system
```

The `system` layer is a generic system interface that chooses an appropriate layer for your file. If the file is on disk, it chooses the `syscall` layer, which maps each user I/O request directly to the corresponding system call. A Fortran `READ` statement is mapped to one or more `read`(2) system calls and a Fortran `WRITE` statement to one or more `write`(2) system calls. This results in almost the same processing as would be done if the `assign -s u` command was used.

If you want your file to be COS blocked, you can specify the following:

```
assign -F cos,system
```

If you want your file to be F77 blocked, you can specify the following:

```
assign -F f77,system
```

These two *spec*s request that each `WRITE` request first be blocked (blocking adds control words to the data in the file to delimit records). The `cos` layer then sends the blocked data to the `system` layer. The `system` layer passes the data to the device.

The process is reversed for `READ` requests. The `system` layer retrieves blocked data from the file. The blocked data is passed to the next higher layer, the `cos` layer, where it is deblocked. The deblocked data is then presented to the user.

I/O Layers

Several different layers are available for the *spec* argument. Each layer invokes one or more layers, which then handles the data it is given in an appropriate manner. For example, the `syscall` layer essentially passes each request to an appropriate system call.

The following tables list the classes you can specify for the *spec* argument to the `assign -F` option:

Table 9-1 Available I/O Layers

Layer	**Function**
`bufa`	Asynchronous buffering layer
`cache`	Memory cached I/O
`cachea`	Asynchronous memory cached I/O

`cos` or `blocked`	COS blocking
`fd`	File descriptor open
`f77`	Record blocking common to most UNIX Fortran implementations
`global`	Distributed cache layer
`null`	Syntactic convenience for users (does nothing)
`site`	Site-specific layer
`syscall`	System call I/O
`system`	Generic system interface
`text`	Newline separated record formats
`tmf`	IRIX tape management facility
`user`	User-written layer

Layered I/O Options

You can modify the behavior of each I/O layer. The following *spec* format shows how you can specify a *class* and one or more *opt* and *num* fields:

class.opt1.opt2:num1:num2:num3

For *class*, you can specify one of the layers listed in the previous tables. Each of the layers has a different set of options and numeric parameter fields that can be specified. This is necessary because each layer performs different duties. The following rules apply to the *spec* argument:

- The *class* and *opt* fields are case-insensitive. For example, the following two *spec*s are identical:

  ```
  Ibm.VBs:100:200

  IBM.vbS:100:200
  ```

- The *opt* and *num* fields are usually optional, but sufficient separators must be specified as placeholders to eliminate ambiguity. For example, the following *spec* s are identical:

  ```
  cos..::40, cos.::40
  cos::40
  ```

In this example, *opt1*, *opt2*, *num1*, and *num2* can assume default values. Similarly, the `sds` layer also allows optional *opt* and *num* fields and it sets *opt1*, *opt2*, *num1*, *num2*, and *num3* to default values as required.

- To specify more than one *spec*, use commas between *specs*. Within each *spec*, you can specify more than one *opt* and *num*. Use periods between *opt* fields, and use colons between *num* fields.

Chapter 10

Using FFIO

This chapter describes how you can use flexible file I/O (FFIO) with common file structures and how to enhance code performance without changing your source code.

FFIO on IRIX systems

The FFIO library calls the `aio_sgi_init` library routine the first time the library issues an asynchronous I/O call. It passes the following parameters to `aio_sgi_init`:

```
aio_numusers=MAX(64,sysconf(_SC_NPROC_CONF))
aio_threads=5
aio_locks=3
```

If a program is using multiple threads and asynchronous I/O, it is important that the value in `aio_numusers` be at least as large as the number of sprocs or pthreads that the application contains. See the `aio_sgi_init` man page for more details.

Users can change these values by setting the following environment variables to the desired value:

- change `FF_IO_AIO_THREADS` to modify `aio_threads`
- change `FF_IO_AIO_LOCKS` to modify `aio_locks`
- change `FF_IO_AIO_NUMUSERS` to modify `aio_numusers`

In the following example, `aio_threads` is set to 8 when the FFIO routines call `aio_sgi_init`:

```
setenv FF_IO_AIO_THREADS 8
```

Users can also supersede the FFIO library's call to `aio_sgi_init` by calling it themselves, before the first I/O statement in their programs.

The following FFIO layers may issue asynchronous I/O calls on IRIX systems:

- `cos`: see the description of `cos` on the `INTRO_FFIO`(3f) man page for a description of the circumstances when the `cos` layer uses asynchronous I/O.

- `cachea` and `bufa`: users should assume that these layers may issue asynchronous I/O calls.
- `system` or `syscall`: these layers may issue asynchronous I/O calls if called from a `BUFFER IN` or `BUFFER OUT` Fortran statement, or if called from one of the listed layers.

FFIO and Common Formats

This section describes the use of FFIO with common file structures and describes the correlation between the common and/or default file structures and the FFIO usage that handles them.

Reading and Writing Text Files

Most human-readable files are in *text format*; this format contains records comprised of ASCII characters with each record terminated by an ASCII line-feed character, which is the newline character in UNIX terminology. The FFIO specification that selects this file structure is `assign -F text`.

The FFIO package is seldom required to handle text files. In the following types of cases, however, using FFIO may be necessary:

- Optimizing text file access to reduce I/O wait time
- Handling multiple EOF records in text files
- Converting data files to and from other formats

I/O speed is important when optimizing text file access. Using `assign -F text` is expensive in terms of CPU time, but it lets you use memory-resident files, which can reduce or eliminate I/O wait time.

The FFIO system also can process text files that have embedded EOF records. The `~e` string alone in a text record is used as an EOF record. Editors such as `sed`(1) or other standard utilities can process these files, but it is sometimes easier with the FFIO system.

Use the `fdcp` command to copy files while converting record blocking.

Reading and Writing Unblocked Files

The simplest form of data file format is the simple binary stream or *unblocked data*. It contains no record marks, file marks, or control words. This is usually the fastest way to move large amounts of data, because it involves a minimal amount of CPU and system overhead.

The FFIO package provides the `syscall` layer, which is designed specifically to handle this binary stream of data. The unblocked binary stream is usually used for unformatted data transfer. It is not usually useful for text files or when record boundaries or backspace operations are required. The complete burden is placed on the application to know the format of the file and the structure and type of the data contained in it.

This lack of structure also allows flexibility; for example, a file declared with one of these layers can be manipulated as a direct-access file with any desired record length.

In this context, `fdcp` can be called to do the equivalent of the `cp`(1) command only if the input file is a binary stream and to remove blocking information only if the output file is a binary stream.

Reading and Writing Fixed-length Records

The most common use for fixed-length record files is for Fortran direct access. Both unformatted and formatted direct-access files use a form of fixed-length records. The simplest way to handle these files with the FFIO system is with binary stream layers, such as `system`, `syscall`, `cache`, and `cachea`. These layers allow any requested pattern of access and also work with direct-access files. The `syscall` and `system` layers, however, are unbuffered and do not give optimal performance for small records.

The FFIO system also directly supports some fixed-length record formats.

Reading and Writing COS Blocked Files

The `cos` layer is provided to sequential unformatted files. It provides for COS blocked files on disk and on magnetic tape and it supports multifile COS blocked datasets.

The `cos` layer must be specified for COS blocked files. If COS is not the default file structure, or if you specify another layer you may have to specify a `cos` layer to get COS blocking.

Enhancing Performance

FFIO can be used to enhance performance in a program without changing the source code or recompiling the code. This section describes some basic techniques used to optimize I/O performance. Additional optimization options are discussed in Chapter 12, "I/O Optimization ", page 77.

Buffer Size Considerations

In the FFIO system, buffering is the responsibility of the individual layers; therefore, you must understand the individual layers in order to control the use and size of buffers.

The `cos` layer has high payoff potential to the user who wants to extract top performance by manipulating buffer sizes. As the following example shows, the `cos` layer accepts a buffer size as the first numeric parameter:

```
assign -F cos:42 u:1
```

The preceding example declares a working buffer size for the `cos` layer of forty-two 4096–byte blocks. This is an excellent size for a file that resides on a DD-49 disk drive because a track on a DD-49 disk drive is comprised of forty-two 4096–byte blocks.

If the buffer is sufficiently large, the `cos` layer also lets you keep an entire file in the buffer and avoid almost all I/O operations.

Removing Blocking

I/O optimization usually consists of reducing overhead. One part of the overhead in doing I/O is the CPU time spent in record blocking. For many files in many programs, this blocking is unnecessary. If this is the case, the FFIO system can be used to deselect record blocking and thus obtain appropriate performance advantages.

The following layers offer unblocked data transfer:

Layer	**Definition**
`syscall`	System call I/O
`bufa`	Buffering layer
`cachea`	Asynchronous cache layer

`cache`	Memory-resident buffer cache

You can use any of these layers alone for any file that does not require the existence of record boundaries. This includes any applications that are written in C that require a byte stream file.

The `syscall` layer offers a simple direct system interface with a minimum of system and library overhead. If requests are larger than approximately 32 Kbytes, this method can be appropriate, especially if the requests are a uniform multiple of 4096 bytes.

The other layers are discussed in the following sections.

The `bufa` and `cachea` Layers

The `bufa` layer and `cachea` layer permits efficient file processing. Both layers provide library-managed asynchronous buffering, and the `cachea` layer allows recently accessed parts of a file to be cached either in main memory or in a secondary data segment.

The number of buffers and the size of each buffer is tunable. In the `bufa`*:bs:nbufs* or `cachea`*:bs:nbufs* FFIO specifications, the *bs* argument specifies the size in 4096–byte blocks of each buffer. The *nbufs* argument specifies the number of buffers to use.

The `cache` Layer

The `cache` layer permits efficient file processing for repeated access to one or more regions of a file. It is a library-managed buffer cache that contains a tunable number of pages of tunable size.

To specify the `cache` layer, use the following option:

```
assign -F cache[:[bs][:[nbufs]]]
```

The *bs* argument specifies the size in 4096–byte blocks of each cache page; the default is 8. The *nbufs* argument specifies the number of cache pages to use. The default is 4. You can achieve improved I/O performance by using one or more of the following strategies:

- Use a cache page size (*bs*) that is a multiple of the disk 4096–byte block or track size. This improves the performance when flushing and filling cache pages.

- Use a cache page size that is a multiple of the user's record size. This ensures that no user record straddles two cache pages. If this is not possible or desirable, it is best to allocate a few additional cache pages *(nbufs).*
- Use a number of cache pages that is greater than or equal to the number of file regions the code accesses at one time.

If the number of regions accessed within a file is known, the number of cache pages can be chosen first. To determine the cache page size, divide the amount of memory to be used by the number of cache pages. For example, suppose a program uses direct access to read 10 vectors from a file and then writes the sum to a different file:

```
integer VECTSIZE, NUMCHUNKS, CHUNKSIZE
parameter(VECTSIZE=1000*512)
parameter(NUMCHUNKS=100)
parameter(CHUNKSIZE=VECTSIZE/NUMCHUNKS)
real*8 a(CHUNKSIZE), sum(CHUNKSIZE)
open(11,access='direct',recl=CHUNKSIZE*8)
call asnunit (2,'-s unblocked',ier)
open (2,form='unformatted')
do i = 1,NUMCHUNKS
  sum = 0.0
  do j = 1,10
    read(11,rec=(j-1)*NUMCHUNKS+i)a
    sum=sum+a
  enddo
  write(2) sum
enddo
end
```

If 4 Mbytes of memory are allocated for buffers for unit 11, 10 cache pages should be used, each of the following size:

```
4MB/10 = 400000 bytes = 97 4096-byte blocks
```

Make the buffer size an even multiple of the record length of 40960 bytes by rounding it up to 100 4096–byte blocks (= 40960 bytes), then use the following `assign` command:

```
assign -F cache:100:10 u:11
```

Foreign File Conversion

This chapter contains information about data conversion, a discussion about moving data between machines, and information about the working of implicit and explicit data conversion. It also explains the support provided for reading and writing files in foreign formats, including the record blocking and numeric and character conversion.

These routines convert data (primarily floating-point data, but also integer and character, as well as Fortran complex and logical data) from your system's native representation to a foreign representation, and vice versa.

Conversion Overview

Data can be transferred between computer systems in several ways. Several formats are supported. For each foreign file type, several supported file and record formats exist or explicit or implicit data conversion can also be used.

When processing foreign data, you must consider the interactions between the data formats and the chosen method of data transfer. This section describes, in broad terms, the techniques available to do these data conversions.

Explicit data conversion is the process by which the user performs calls to subroutines that convert the *native* data to and from the *foreign* data formats. These routines are provided for many data formats. This is discussed in more detail in "Explicit Data Item Conversion", page 68.

Implicit data conversion is the process by which users declare that a particular file contains foreign data and/or record blocking and then request that the run-time library perform appropriate transformations on the data to make it useful to the program at I/O time. This method of record and/or data format conversion requires changes in command scripts. This is discussed in more detail in "Implicit Data Item Conversion", page 69.

Using `fdcp` to Transfer Files

The `fdcp`(1) command can handle data that is not a simple disk-resident byte stream. The `fdcp` command assumes that both the data and any record, including EOF records, can be copied from one file to another. Record structures can be preserved or removed. EOF records can be preserved either as EOF records in the output file or used to separate the delimited data in the input file into separate files.

The `fdcp` command does not perform data conversion; the only transformations done are on the record and file structures (`fdcp` transforms block, record, and file control words from one format to another).

If no `assign`(1) information is available for a file, the `system` layer is used. This means that if the file being accessed is on disk and if no `assign -F` attribute is used, the `syscall` layer is used; if it is on a tape, the `bmx` layer is used. Therefore, each tape block is considered a record; user tape marks are mapped to EOF.

Data Item Conversion

Both implicit and explicit conversion of data items are provided. Explicit conversion means that the user's code must invoke the routines that convert between native systems and foreign representations.

Options to the `assign`(1) command control implicit conversion. The data types in the Fortran I/O lists direct implicit conversion. Implicit conversion is usually transparent to users and is available only to Fortran programmers. The following sections describe these data conversion types and provide direction in choosing a conversion type.

Explicit Data Item Conversion

The Fortran library contains a set of subroutines that convert between data formats of various vendors. These routines are callable from any supported programming language. For complete details, see the individual man pages for each routine. These subroutines provide an efficient way to convert data that was read into system central memory.

The following table lists these conversion routines.

Table 11-1 Available conversion routines

Non-IEEE	`CRY2MIPS`	`MIPS2CRY`
IEEE Fortran conversion	`IEG2MIPS`	`MIPS2IEG`
VAX Fortran conversion	`VAX2MIPS`	`MIPS2VAX`

See the individual man pages for details about the syntax and arguments for each routine.

Implicit Data Item Conversion

Implicit data conversion in Fortran requires no explicit action by the program to convert the data in the I/O stream other than using the `assign` command to instruct the libraries to perform conversion. For details, see the `assign`(1) man page.

The implicit data conversion process is performed in two steps:

1. Record format conversion
2. Data conversion

Record format conversion interprets or converts the internal record blocking structures in the data stream to gain record-level access to the data. The data contained in the records can then be converted.

Using implicit conversion, you can select record blocking or deblocking alone, or you can request that the data items be converted automatically. When enabled, record format conversion and data item conversion occur transparently and simultaneously. Changes are usually not required in your Fortran code.

To enable conversion of foreign record formats, specify the appropriate record type with the `assign -F` command. The `-N` (numeric conversion) and `-C` (character conversion) `assign` options control conversion of data contained in a record. If `-F` is specified, but `-N` and `-C` are not, the libraries interpret the record format, but they do not convert data. You can obtain information about the type of data that will be converted (and, therefore, the type of conversion that will be performed) from the Fortran I/O list.

If `-N` is used and `-C` is not, an appropriate character conversion type is selected by default, as shown in the following table.

Table 11-2 Conversion types

`-N` option	`-C` default	Meaning
none	none	No data conversion
default	default	No data conversion
`cray`	ASCII	Non-IEEE data conversion
`mips`	ASCII	No data conversion
`user`	ASCII	User defined data conversion
`site`	ASCII	Site defined data conversion
`ieee`	ASCII	Generic 32–bit IEEE data conversion
`ieee_32`		(alias for above)
`ieee_64`	ASCII	Cray 64–bit IEEE data conversion
`ieee_le`	ASCII	Little-endian 32–bit IEEE data conversion
`vax`	ASCII	DEC VAX/VMS data conversion
`vms`		(alias for above)

Supported implicit data conversion includes conversion of the supported tape and disk formats and data types through standard Fortran formatted, unformatted list-directed, and Namelist I/O and through `BUFFER IN` and `BUFFER OUT` statements. Generally, read, write, and rewind are supported for all record formats.

If you select the `-N` option, the libraries perform data conversion for Fortran unformatted statements and `BUFFER IN` and `BUFFER OUT` I/O statements. Data is converted according to its Fortran data type. Table 11-3, page 71 describes the conversion performed for each of the conversion types.

For numeric data conversions, most foreign data elements are defined with fewer bits than their corresponding native data elements. If the value in a native element is too large to fit in the foreign element, the foreign element is set to the largest or smallest possible value; no error is generated. When converting from a native element to a smaller foreign element, precision is also lost due to truncation of the floating-point mantissa.

If the `assign -N user` or `assign -N site` command is specified, the user or site must provide `site` numeric data conversion routines. They follow the same calling conventions as the other explicit routines.

Table 11-3 Supported foreign I/O formats and default data types

Vendor data type	Record formats	Foreign data types	Native data types

IBM	`U`, `F`, `FB`, `V`, `VB`, `VBS`	`INTEGER*2`
		`INTEGER*4`
		`DOUBLE PRECISION`
		`COMPLEX*4`
		`LOGICAL*4`
		`CHARACTER` (EBCDIC)
			`INTEGER(24/32)`
			`INTEGER(64)`
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

VMS	`F`, `V`, `S` for tape; `bb` or `disk` and `tr` types	`INTEGER*2`
		`INTEGER*4`
		`REAL*4`
		`DOUBLE PRECISION`
		`COMPLEX*4`
		`LOGICAL*4`
		`CHARACTER` (ASCII)
			`INTEGER(24/32)`
			`INTEGER(64)`
			`REAL(64)`
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

CDC (60 bit)	Subtype: `DISK`, `I`, `SI` Block record: `IW`, `CW`, `CZ`, `CS`	`INTEGER`
		`REAL`
		`DOUBLE PRECISION`
		`COMPLEX`
		`LOGICAL`
		`CHARACTER` (display code)
			`INTEGER`
			`REAL`
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

CDC NOS/VE	`F`, `S`, `V`	`INTEGER`
		`REAL`
		`DOUBLE PRECISION`
		`COMPLEX`
		`LOGICAL`
		`CHARACTER`
			`INTEGER`
			`REAL`
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

Vendor data type	Record formats	Foreign data types	Native data types

CDC/ETA CYBER205	`W` type	`INTEGER`
		`REAL`
		`REAL*4`
		`DOUBLE PRECISION`
		`COMPLEX`
		`LOGICAL`
		`CHARACTER` (display code)
			`INTEGER`
			`REAL`
			`INTEGER(24/32)` (See Note 1)
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

IEEE	None defined (often f77)	`INTEGER*2` (see Note 2)
		`INTEGER*4`
		`REAL*4`
		`DOUBLE PRECISION`
		`COMPLEX*4`
		`LOGICAL*4`
		`CHARACTER` (ASCII)
			`INTEGER(24/32)`
			`INTEGER(64)`
			`REAL(64)`
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

ULTRIX	`f77.vax`	`INTEGER*2`
		`INTEGER*4`
		`REAL*4`
		`DOUBLE PRECISION`
		`COMPLEX*4`
		`LOGICAL*4`
		`CHARACTER` (ASCII)
			`INTEGER(24/32)`
			`INTEGER(64)`
			`REAL(64)` (see Note 3)
			`DOUBLE PRECISION`
			`COMPLEX`
			`LOGICAL`
			`CHARACTER` (ASCII)

Note 1: The CYBER 205 half-precision type maps to the short integer (`INTEGER*2`) type

For implicit conversion, specify format characteristics on an `assign` command.

Files can be converted to one of the following:

- A magnetic tape
- A disk file
- A file transferred from a front end with the station

When a Fortran I/O operation is performed on the file, the appropriate file format and data conversions are performed during the I/O operation. Data conversion is performed on each data item, based on the type of the Fortran variable in the I/O list.

For example, if the first read of a foreign format file is the following, the library interprets any blocking structures in the file that precede the first data record:

```
READ (10) INT,FLOAT1,FLOAT2
```

These vary depending on the file type and record format. The first 32 bits of data (in IBM format, for example) are extracted, sign-extended, and stored in the `INT` Fortran variable. The next 32 bits are extracted, converted to native floating-point format, and stored in the `FLOAT1` Fortran variable.

The next 32 bits are extracted, converted, and stored into the `FLOAT2` Fortran variable. The library then skips to the end of the foreign logical record. When writing from a native system to a foreign format (for example, if in the previous example `WRITE(10)` was used), precision is lost when converting from a 64-bit representation to 32-bit representation.

Choosing a Conversion Method

As with any software process, the various options for data conversion have advantages and disadvantages, which are discussed in this section. As a set, various data conversion options provide choices in methods of file processing for front-end systems. No one option is best for all applications.

Explicit Conversion

Explicit data conversion has some distinct advantages over using station software, including the following:

- Direct control over data conversion is provided (including some options not available through implicit conversion).
- Programmers can control the conversion, and they can do the conversion at a convenient and appropriate time.
- Conversion is usually performed on large data areas as vector operations, increasing performance.

One disadvantage of using explicit conversion is that explicit routines require changes to the source code.

Implicit Conversion

An advantage when using implicit conversion is that you do not have to change the source code.

The following are disadvantages when using implicit conversion:

- Job Control Language (JCL) or script changes are required on the `assign`(1) command.
- Conversion is less efficient on a record-by-record basis.
- Conversion is done at I/O time according to the declared data types, allowing little flexibility for nonstandard requirements.

Foreign Conversion Techniques

This section contains some tips and techniques for the following conversion types:

Conversion type	**Convert data to/from**
IEEE conversion	Various types of workstations and different vendors that support IEEE floating-point format
VAX/VMS conversion	DEC VAX machines that run MVS

Workstation and IEEE Conversion

IRIX systems use 32-bit IEEE standard floating point, as do many workstations and personal computers. These workstations often use a dialect of UNIX software as the operating system, with twos-complement arithmetic and the ASCII character set. The logical values in these implementations are usually the same for Fortran and C. They use zero for false and nonzero for true. It is also common to see the `f77` record blocking used by the Fortran run-time library on unformatted sequential files.

No IEEE record format exists, but the IEEE implicit and explicit data conversion routine facilities are provided with the assumption that many of these things are true.

Most computer systems that use the IEEE data formats run operating systems based on UNIX software and use `f77` record blocking. You can use the `rcp` or `ftp` commands to transfer files. In most cases, the following command should work for implicit conversion:

```
assign -F f77 -N ieee fort.1
```

When writing files in the `f77` format, remember that you can gain a large performance boost by ensuring that the records being written fit in the working buffer of the `f77` layer.

SGI MIPS systems use IEEE floating-point representation, so IEEE conversion is usually unnecessary when reading or writing IEEE data on these systems.

On MIPS systems, data types can be declared as 64-bits in size and can then be read or written directly. This is the most direct and efficient method to read or write data files for IEEE systems. The user can either alter the declarations of the variables used in the Fortran I/O list to declare them as `KIND=8` or as `REAL*8` (or `INTEGER*8`), or all the variables in the program can be resized by compiling with the `-r8` (or `-i8`) compiler option.

The following are other IEEE data conversion variants; not all variants are available on all systems:

`ieee` or `ieee_32`	The default workstation conversion specification. Data sizes are based on 32-bit words.
`ieee_64`	Data sizes are based on 64-bit words.
`ieee_dp`	Data sizes are based on 32-bit words except for floating-point data which is based on 64-bit words.
`ieee_le` or `ultrix`	Data sizes are based on 32-bit words and are little-endian.
`ieee_le_dp` or `ultrix_dp`	Data sizes are based on 32-bit words except for floating-point data which is based on 64-bit words. All data is little-endian.
`mips`	Data sizes are based on 32-bit words except for 128-bit floating-point data which uses a "double double" format.

Chapter 12

I/O Optimization

Although I/O performance is one of the strengths of supercomputers, speeding up the I/O in a program is an often neglected area of optimization. A small optimization effort can often produce a surprisingly large gain.

The run-time I/O library contains low overhead, built-in instrumentation that can collect vital statistics on activities such as I/O. This run-time library, together with `procstat`(1) and other related commands, offers a powerful tool set that can analyze the program I/O without accessing the program source code.

A wide selection of optimization techniques are available through the flexible file I/O (FFIO) system. You can use the `assign`(1) command to invoke FFIO for these optimization techniques. This chapter stresses the use of `assign` and FFIO because these optimization techniques do not require program recompilation or relinking.

This chapter describes ways to identify code that can be optimized and the techniques that you can use to optimize the code.

Overview

I/O can be represented as a series of *layers* of data movement. Each layer involves some processing. Figure 12-1, page 78 shows typical output flow from the system to disk.

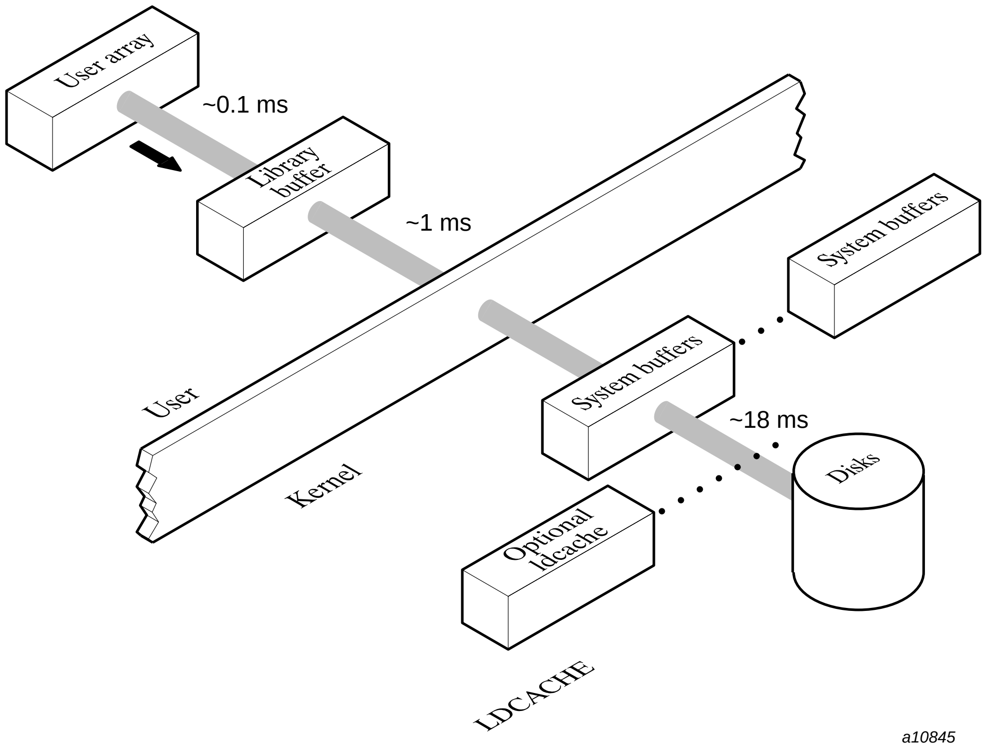


Figure 12-1 I/O layers

On output, data moves from the user space to a library buffer, where small chunks of data are collected into larger, more efficient chunks. When the library buffer is full, a system request is made and the kernel moves the data to a system buffer. From there, the data is sent through the I/O processor (IOP), perhaps through ldcache, to the device. On input, the path is reversed.

The times shown in Figure 12-1 may not be duplicated on your system because many variables exist that affect timing. These times do, however, give an indication of the times involved in each processing stage.

For optimization purposes, it is useful to differentiate between permanent files and temporary files. *Permanent files* are external files that must be retained after the program completes execution. *Temporary files* or *scratch files* are usually created and reused during the execution of the program, but they do not need to be retained at the end of the execution.

Permanent files must be stored on actual devices. Temporary files exist in memory and do not have to be written to a physical device. With temporary files, the strategy is to avoid using system calls (going to "lower layers" of I/O processing). If a temporary file is small enough to reside completely in memory, you can avoid using system calls.

Permanent files require system calls to the kernel; because of this, optimizing the I/O for permanent files is more complicated. I/O on permanent files may require the full complement of I/O layers. The goal of I/O optimization is to move data to and from the devices as quickly as possible. If that is not fast enough, you must find ways to overlap I/O with computation.

An Overview of Optimization Techniques

This section briefly describes the optimization techniques that are discussed in the remainder of this chapter.

Optimizations Not Affecting Source Code

The following types of optimization may improve I/O performance:

- Specify the cache page size so that one or more records will fit on a cache page if the program is using unformatted direct access I/O (see "Using a Cache Layer", page 81, for details).
- Use file structures without record control information to bypass the overhead associated with records (see "Using Simpler File Structures", page 84, for details).
- Choose file processing with appropriate buffering strategies. The `cos`, `bufa`, and `cachea` FFIO layers implement asynchronous write-behind (see "Using Asynchronous Read-ahead and Write-behind", page 83, for details). The `cos` and `bufa` FFIO layers implement asynchronous read-ahead; this is available for the `cachea` layer through use of an `assign` option.
- Choose efficient library buffer sizes. Bypass the library buffers when possible by using the `system` or `syscall` layers (see "Changing Library Buffer Sizes", page 85, for details).

- Use the `assign` command to specify scratch files to prevent writes to disk and to delete the files when they are closed (see "Scratch Files", page 82, for details).

"Enhancing Performance", page 64, also provides further information about using FFIO to enhance I/O performance.

Optimizations That Affect Source Code

The following source program changes may affect the I/O performance of a Fortran program:

- Use unformatted I/O when possible to bypass conversion of data.
- Use whole array references in I/O lists where possible. The generated code passes the entire array to the I/O library as the I/O list item rather than pass it through several calls to the I/O library.
- Use special packages such as buffer I/O, random-access I/O, and asynchronous queued I/O.
- Overlap CPU time and I/O time by using asynchronous I/O.

Optimizing I/O Speed

I/O optimization can often be accomplished by simply addressing I/O speed. The following storage systems are available, ranked in order of speed:

- CPU main memory
- Magnetic disk drives
- Optional magnetic tape drives

Fast storage systems are expensive and have smaller capacities. You can specify a fast device through FFIO layers and use several FFIO layers to gain the maximum performance benefit from each storage medium. The remainder of this chapter discusses many of these FFIO optimizations. These easy optimizations are frequently those that yield the highest payoffs.

Optimizing System Requests

In a busy interactive environment, queuing for service is time consuming. In tuning I/O, the first step is to reduce the number of physical delays and the queuing that results by reducing the number of system requests, especially the number of system requests that require physical device activity.

System requests are made by the library to the kernel. They request data to be moved between I/O devices. Physical device activity consumes the most time of all I/O activities.

Typical requests are read, write, and seek. These requests may require physical device I/O. During physical device I/O, time is spent in the following activities:

- Transferring data between disk and memory.
- Waiting for physical operations to complete. For example, moving a disk head to the cylinder (seek time) and then waiting for the right 4096–byte block to come under the disk head (latency time).

System requests can require substantial CPU time to complete. The system may suspend the requesting job until a relatively slow device completes a service.

Besides the time required to perform a request, the potential for congestion also exists. The system waits for competing requests for kernel, disk, IOP, or channel services. System calls to the kernel can slow I/O by one or two orders of magnitude.

The information in this section summarizes some ways you can optimize system requests.

Using a Cache Layer

The FFIO `cache` layer keeps recently used data in fixed size main memory or *cache pages* in order to reuse the data directly from these buffers in subsequent references. It can be tuned by selecting the number of cache pages and the size of these pages.

The use of the `cache` layer is especially effective when access to a file is localized to some regions of the whole file. Well-tuned cached I/O can be an order of magnitude faster than the default I/O.

Even when access is sequential, the `cache` layer can improve the I/O performance. For good performance, use page sizes large enough to hold the largest records.

The `cache` layers work with the standard Fortran I/O types and the compiler extensions of `BUFFER IN/OUT`, `READMS/WRITMS`, and `GETWA/PUTWA`.

The following `assign` command requests 100 pages of 42 blocks each:

```
assign -F cache:42:100 f:filename
```

Specifying cache pages of 42 blocks matches the track size of a DD-49 disk.

Optimizing File Structure Overhead

The Fortran standard uses the *record* concept to govern I/O. It allows you to skip to the next record after reading only part of a record, and you can backspace to a previous record. The I/O library implements Fortran records by maintaining an internal record structure.

In the case of a sequential unformatted file, it uses a COS blocked file structure, which contains control information that helps to delimit records. The I/O library inserts this control information on write operations and removes the information on read operations. This process is known as *record translation*, and it consumes time.

If the I/O performed on a file does not require this file structure, you can avoid using the blocked structure and record translation. However, if you must do positioning in the file, you cannot avoid using the blocked structure.

The information in this section describes ways to optimize your file structure overhead.

Scratch Files

Scratch files are temporary and are deleted when they are closed. To decrease I/O time, move applications' scratch files from user file systems to high-speed file systems.

When optimizing, you should avoid writing the data to disk. This is especially important if most of the data can be held in main memory.

Fortran lets you open a file with `STATUS='SCRATCH'`. It also lets you close temporary files by using a `STATUS='DELETE'`. These files are placed on disk, unless the `.scr` specification for FFIO or the `assign -t` command is specified for the file. Files specified as `assign -t` or `.scr` are deleted when they are closed.

Using Asynchronous Read-ahead and Write-behind

Several FFIO layers automatically enhance I/O performance by performing asynchronous read-ahead and write-behind. These layers include:

- `cos`: default Fortran sequential unformatted file. Specified by `assign -F cos`.
- `bufa`: specified by `assign -F bufa`.
- `cachea`: default Fortran direct unformatted files. Specified by `assign -F cachea`. Default `cachea` behavior provides asynchronous write-behind. Asynchronous read-ahead is not enabled by default, but is available by an `assign` option.

If records are accessed sequentially, the `cos` and `bufa` layers will automatically and asynchronously pre-read data ahead of the file position currently being accessed. This behavior can be obtained with the `cachea` layer with an `assign` option; in that case, the `cachea` layer will also detect sequential backward access patterns and pre-read in the reverse direction.

Many user codes access the majority of file records sequentially, even with `ACCESS='DIRECT'` specified. Asynchronous buffering provides maximum performance when:

- Access is mainly sequential, but the working area of the file cannot fit in a buffer or is not reused frequently.
- Significant CPU-intensive processing can be overlapped with the asynchronous I/O.

Use of automatic read-ahead and write-behind may decrease execution time by half because I/O and CPU processing occur in parallel.

The following `assign` command specifies a specific `cachea` layer with 10 pages, each the size of a DD-40 track. Three pages of asynchronous read-ahead are requested. The read-ahead is performed when a sequential read access pattern is detected.

```
assign -F cachea:48:10:3 f:filename
```

This command would work for a direct access or sequential Fortran file which has unblocked file structure.

Using Simpler File Structures

Marking records incurs overhead. If a program reads all of the data in any record it accesses and avoids the use of `BACKSPACE`, you can make some minor performance savings by eliminating the overhead associated with records. This can be done in several ways, depending on the type of I/O and certain other characteristics.

For example, the following `assign` statements specify the unblocked file structure:

```
assign -s unblocked f:filename
assign -s u f:filename
assign -s bin f:filename
```

Minimizing Data Conversions

When possible, avoid formatted I/O. Unformatted I/O is faster, and it avoids potential inaccuracies due to conversion. Formatted Fortran I/O requires that the library interpret the `FORMAT` statement and then convert the data from an internal representation to ASCII characters. Because this must be done for every item generated, it can be very time-consuming for large amounts of data.

Whenever possible, use unformatted I/O to avoid this overhead. Do not use edit-directed I/O on scratch files. Major performance gains are possible.

You can explicitly request data conversions during I/O. The most common conversion is through Fortran edit-directed I/O. I/O statements using a `FORMAT` statement, list-directed I/O, and namelist I/O require data conversions.

Conversion between internal representation and ASCII characters is time-consuming because it must be performed for each data item. When present, the `FORMAT` statement must be parsed or interpreted. For example, it is very slow to convert a decimal representation of a floating-point number specified by an `E` edit descriptor to an internal binary representation of that number.

For more information about data conversions, see Chapter 11, "Foreign File Conversion", page 67.

Minimizing Data Copying

The Fortran I/O libraries usually use main memory buffers to hold data that will be written to disk or was read from disk. The library tries to do I/O efficiently on a few large requests rather than in many small requests. This process is called *buffering*.

Overhead is incurred and time is spent whenever data is copied from one place to another. This happens when data is moved from user space to a library buffer and when data is moved between buffers. Minimizing buffer movement can help improve I/O performance.

Changing Library Buffer Sizes

The libraries generally have default buffer sizes. The default is suitable for many devices, but major performance improvements can result from requesting an efficient buffer size.

The optimal buffer size for very large files is usually a multiple of a device allocation for the disk. This may be the size of a track on the disk. If optimal size buffers are used and the file is contiguous, disk operations are very efficient. Smaller sizes require more than one operation to access all of the information on the allocation or track. Performance does not improve much with buffers larger than the optimal size, unless striping is specified.

When enough main memory is available to hold the entire file, the buffer size can be selected to be as large as the file for maximum performance.

The maximum length of a formatted record depends on the size of the buffer that the I/O library uses for a file. The size of the buffer depends on the following:

- hardware system and OS level
- Type of file (external or internal)
- Type of access (sequential or direct)
- Type of formatted I/O (edit-directed, list-directed, or namelist)

Bypassing Library Buffers

After a request is made, the library usually copies data between its own buffers and the user data area. For small requests, this may result in the blocking of many requests into fewer system requests, but for large requests when blocking is not needed, this is inefficient. You can achieve performance gains by bypassing the library buffers and making system requests to the user data directly.

To bypass the library buffers and to specify a direct system interface, use the `assign -s u` option or specify the FFIO `system`, or `syscall` layer, as is shown in the following `assign` command examples:

```
assign  -s u  f:filename
assign  -F system  f:filename
assign  -F syscall  f:filename
```

The user data should be in multiples of the disk sector size (usually 4096 bytes) for best disk I/O performance.

If library buffers are bypassed, the user data should be on a 4096–byte boundary to prevent I/O performance degradation.

Other Optimization Options

There are other optimizations that involve changing your program. The following sections describe these optimization techniques.

Using Pipes

When a program produces a large amount of output used only as input to another program consider using pipes. If both programs can run simultaneously, data can flow directly from one to the next by using a pipe. It is unnecessary to write the data to the disk. See Chapter 4, "Named Pipe Support ", page 21, for details about pipes.

Overlapping CPU and I/O

Major performance improvements can result from overlapping CPU work and I/O work. This approach can be used in many high-volume applications; it simultaneously uses as many independent devices as possible.

To use this method, start some I/O operations and then immediately begin computational work without waiting for the I/O operations to complete. When the computational work completes, check on the I/O operations; if they are not completed yet, you must wait. To repeat this cycle, start more I/O and begin more computations.

As an example, assume that you must compute a large matrix. Instead of computing the entire matrix and then writing it out, a better approach is to compute one column at a time and to initiate the output of each column immediately after the column is computed. An example of this follows:

```
     dimension a(1000,2000)
     do 20 jcol= 1,2000
       do 10 i= 1,1000
         a(i,jcol)= sqrt(exp(ranf()))
10     continue
20   continue
     write(1) a
     end
```

First, try using the `assign -F cos.async f:filename` command. If this is not fast enough, rewrite the previous program to overlap I/O with CPU work, as follows:

```
      dimension a(1000,2000)
      do 20 jcol= 1,2000
        do 10 i= 1,1000
          a(i,jcol)= sqrt(exp(ranf()))
10      continue
        BUFFER OUT(1,0) (a(1,jcol),a(1000,jcol) )
20    continue
      end
```

The following Fortran statements and library routines can return control to the user after initiating I/O without requiring the I/O to complete:

- `BUFFER IN` and `BUFFER OUT` statements (buffer I/O)
- FFIO `cos` blocking asynchronous layer (available on IRIX systems)
- FFIO `cachea` layer (available on IRIX systems)
- FFIO `bufa` layer (available on IRIX systems)

FFIO Layer Reference

This chapter provides details about each of the following FFIO layers:

Layer	**Definition**
`bufa`	Library-managed asynchronous buffering
`cache`	`cache` layer
`cachea`	`cachea` layer
`cos`	COS blocking
`event`	I/O monitoring
`f77`	UNIX record blocking
`fd`	File descriptor
`global`	Cache distribution layer
`null`	The `null` layer
`syscall`	System call I/O
`system`	Generic system layer
`text`	Newline separated record formats
`user` and `site`	Writable layer
`vms`	VAX/VMS file formats

Characteristics of Layers

In the descriptions of the layers that follow, the data manipulation tables use the following categories of characteristics:

Characteristic	Description
Granularity	Indicates the smallest amount of data that the layer can handle. For example, layers can read and write a single bit; other layers, such as the `syscall` layer, can process only 8-bit bytes. Still others, such as some CDC formats, process data in units of 6-bit characters in which any operation that is not a multiple of 6 bits results in an error.
Data model	Indicates the data model. Three main data models are discussed in this section. The first type is the `record` model, which has data with record boundaries, and may have an end-of-file (EOF). The second type is `stream` (a stream of bits). None of these support the EOF. The third type is the `filter`, which does not have a data model of its own, but derives it from the lower-level layers. Filters usually perform a data transformation (such as blank compression or expansion).
Truncate on write	Indicates whether the layer forces an implied EOD on every write operation (EOD implies truncation).
Implementation strategy	Describes the internal routines that are used to implement the layer. The X-record type referred to under implementation strategy refers to a record type in which the length of the record is prepended and appended to the record. For `f77` files, the record length is contained in 4 bytes at the beginning and the end of a record. The `v` type of NOS/VE and the `w` type of CYBER 205/ETA also prepend and append the length of the record to the record.

In the descriptions of the layers, the supported operations tables use the following categories:

Operation	Lists the operations that apply to that particular layer. The following is a list of supported operations:

```
ffopen              ffclose
ffread              ffflush
ffreada             ffweof
ffreadc             ffweod
ffwrite             ffseek
ffwritea            ffpos
ffwritec            ffbksp
```

Support	Uses three potential values: Yes, No, or Passed through. "Passed through" indicates that the layer does not directly support the operation, but relies on the lower-level layers to support it.
Used	Lists two values: Yes or No. "Yes" indicates that the operation is required of the next lower-level layer. "No" indicates that the operation is never required of the lower-level layer. Some operations are not directly required, but are passed through to the lower-layer if requested of this layer. These are noted in the comments.
Comments	Describes the function or support of the layer's function.

On many layers, you can also specify the numeric parameters by using a keyword. See the `INTRO_FFIO`(3f) man page for more details about FFIO layers.

When using direct access files, you must assign the file to either the `system` or the `global` layer for code that works with more than one processor. The default layer for direct access is the `cache` layer and it does not have the coherency to handle multiple processes doing I/O to the same file.

Individual Layers

The remaining sections in this chapter describe the individual FFIO layers in more detail.

The bufa Layer

The `bufa` layer provides library-managed asynchronous buffering. This can reduce the number of low-level I/O requests for some files. The syntax is as follows:

```
bufa:[num1]:[num2]
```

The keyword syntax is as follows:

```
bufa[.bufsize=num1][.num_buffers=num2]
```

The *num1* argument specifies the size, in 4096-byte blocks, of each buffer. The default buffer size depends on the device where your file is located. The maximum allowed value for *num1* on IRIX systems is 32,767.

The *num2* argument specifies the number of buffers. The default is 2.

Table 13-1 Data manipulation: `bufa` layer

Granularity	Data model	Truncate on write
8 bits	Stream	No

Table 13-2 Supported operations: `bufa` layer

	Supported operations		Required of next lower level?	
`ffopen`	Yes		Yes	
`ffread`	Yes		Yes	
`ffreada`	Yes	Always synchronous	Yes	
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	
`ffwritea`	Yes	Always synchronous	Yes	

	Supported operations		Required of next lower level?	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		Yes	
`ffweof`	Passed through		Yes	Only if explicitly requested
`ffweod`	Yes		Yes	
`ffseek`	Yes	Only if supported by the underlying layer	Yes	Only if explicitly requested
`ffpos`	Yes		Yes	Only if explicitly requested
`ffbksp`	No		No	

The cache Layer

The `cache` layer allows recently accessed parts of a file to be cached either in main memory. This can significantly reduce the number of low-level I/O requests for some files that are accessed randomly. This layer also offers efficient sequential access when a buffered, unblocked file is needed. The syntax is as follows:

```
cache[.type]:[num1]:[num2][num3]
```

The following is the keyword specification:

```
cache[.type][.page_size=num1][.num_pages=num2
   [.bypass_size=num3]]
```

The *type* argument must be `mem`; this directs that cache pages reside in main memory. *num1* specifies the size, in 4096–byte blocks, of each cache page buffer. The default is 8. The maximum allowed value for *num1* is 32,767.

num2 specifies the number of cache pages. The default is 4. *num3* is the size in 4096–byte blocks at which the `cache` layer attempts to bypass `cache` layer buffering. If a user's I/O request is larger than *num3*, the request might not be copied to a cache page. The default size for *num3* is *num3*=*num1*.

When a cache page must be preempted to allocate a page to the currently accessed part of a file, the least recently accessed page is chosen for preemption. Every access stores a time stamp with the accessed page so that the least recently accessed page can be found at any time.

Table 13-3 Data manipulation: `cache` layer

Granularity	Data model	Truncate on write
8 bit	Stream	No
512 words (cache.sds)	Stream	No

Table 13-4 Supported operations: `cache` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		No	
`ffreada`	Yes	Always synchronous	Yes	
`ffreadc`	Yes		No	
`ffwrite`	Yes		No	
`ffwritea`	Yes	Always synchronous	Yes	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	
`ffweof`	No		No	
`ffweod`	Yes		Yes	
`ffseek`	Yes		Yes	Requires underlying interface to be a stream

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffpos`	Yes		NA	
`ffbksp`	No		NA	

The cachea Layer

The `cachea` layer allows recently accessed parts of a file to be cached either in main memory. This can significantly reduce the number of low-level I/O requests for some files that are accessed randomly.

This layer can provide high write performance by asynchronously writing out selective cache pages. It can also provide high read performance by detecting sequential read access, both forward and backward. When sequential access is detected and when read-ahead is chosen, file page reads are anticipated and issued asynchronously in the direction of file access. The syntax is as follows:

```
cachea[mem]:[num1]:[num2]:[num3]:[num4]
```

The keyword syntax is as follows:

```
cachea[mem][.page_size=num1][.num_pages=num2]
   [.max_lead=num3][.shared_cache=num4]
```

`mem`	Directs that cache pages reside in main memory.
num1	Specifies the size, in 4096-byte blocks, of each cache page buffer. Default is 8. The maximum allowed value for *num1* is 32,767.
num2	Specifies the number of cache pages to be used. Default is 4.
num3	Specifies the number of cache pages to asynchronously read ahead when sequential read access patterns are detected. Default is 0.
num4	Specifies a cache number in the range of 1 to 15. Cache number 0 is a cache which is private to the current FFIO layer. Any cache number

larger than 0 is shared with any other file using a `cachea` layer with the same number.

Multiple `cachea` layers in a chain may not contain the same nonzero cache number.

Stacked shared `cachea` layers are not supported.

The following examples demonstrate this functionality:

- The following specifications cannot both be used by a multitasked program:

```
assign -F cachea::::1,cachea::::2 u:1
assign -F cachea::::2,cachea::::1 u:2
```

Table 13-5 Data manipulation: `cachea` layer

Granularity	Data model	Truncate on write
8 bit	Stream	No

Table 13-6 Supported operations: `cachea` layer

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
`ffopen`	Yes		Yes	
`ffread`	Yes		No	
`ffreada`	Yes		Yes	
`ffreadc`	Yes		No	
`ffwrite`	Yes		No	
`ffwritea`	Yes		Yes	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
`ffweof`	No		No	
`ffweod`	Yes		Yes	
`ffseek`	Yes		Yes	Requires that the underlying interface be a stream
`ffpos`	Yes		NA	
`ffbksp`	No		NA	

The cos Blocking Layer

The `cos` layer performs COS blocking and deblocking on a stream of data. The general format of the `cos` specification follows:

```
cos:[.type][.num1]
```

The format of the keyword specification follows:

```
cos[.type][.bufsize=num1]
```

The *num1* argument specifies the working buffer size in 4096-byte blocks.

If not specified, the default buffer size is the larger of the following: the preferred I/O block size (see the `stat`(2) man page for details), or 8 4096–byte blocks. See the `INTRO_FFIO`(3f) man page for more details.

Reads are always performed in partial read mode; therefore, you do not have to know the block size of a tape to read it (if the tape block size is larger than the buffer, partial mode reads ensure that no parts of the tape blocks are skipped).

Table 13-7 Data manipulation: `cos` layer

Granularity	Data model	Truncate on write	Implementation strategy
1 bit	Records with multi-EOF capability	Yes	`cos` specific

Table 13-8 Supported operations: `cos` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		Yes	
`ffreada`	Yes	Always synchronous	Yes	
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	
`ffwritea`	Yes	Always synchronous	Yes	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes	No-op	Yes	
`ffweof`	Yes		No	
`ffweod`	Yes		Yes	Truncation occurs only on close
`ffseek`	Yes	Minimal support (see following note)	Yes	
`ffpos`	Yes		NA	
`ffbksp`	Yes	No records	No	

The `event` Layer

The `event` layer monitors I/O activity (on a per-file basis) which occurs between two I/O layers. It generates statistics as an ASCII log file and reports information such as the number of times an event was called, the event wait time, the number of bytes requested, and so on. You can request the following types of statistics:

- A list of all event types
- Event types that occur at least once
- A single line summary of activities that shows information such as amount of data transferred and the data transfer rate.

Statistics are reported to `stderr` by default. The `FF_IO_LOGFILE` environment variable can be used to name a file to which statistics are written by the `event` layer. The default action is to overwrite the existing statistics file if it exists. You can append reports to the existing file by specifying a plus sign (+) before the file name, as in this example:

```
setenv FF_IO_LOGFILE +saveIO
```

This layer report counts for `read`, `reada`, `write`, and `writea`. These counts represent the number of calls made to an FFIO layer entry point. In some cases, the `system` layer may actually use a different I/O system call, or multiple system calls. For example, the `reada` system call does not exist on IRIX systems, and the `system` layer `reada` entry point will use `aio_read()`.

Amention of the `lock` layer may be included during report generation even though that layer may not have been specified by the user.

The `event` layer is enabled by default and is included in the executable file; you do not have to relink to study the I/O performance of your program. To obtain event statistics, rerun your program with the `event` layer specified on the `assign` command, as in this example:

```
assign -F bufa, event, cachea, event, system
```

The syntax for the `event` layer is as follows:

```
event[.type]
```

There is no alternate keyword specification for this layer.

The *type* argument selects the level of performance information to be written to the ASCII log file; it can have one of the following values:

Value	**Definition**
`nostat`	No statistical information is reported.
`summary`	Event types that occur at least once are reported.
`brief`	A one line summary for layer activities is reported.

The `f77` Layer

The `f77` layer handles blocking and deblocking of the `f77` record type, which is common to most UNIX Fortran implementations. The syntax for this layer is as follows:

```
f77[.type]:[num1]:[num2]
```

The following is the syntax of the keyword specification:

```
f77[.type][.recsize=num1][.bufsize=num2]
```

The *type* argument specifies the record type and can take one of the following two values:

Value	**Definition**
`nonvax`	Control words in a format common to large machines such as the MC68000; default.
`vax`	VAX format (byte-swapped) control words.

The *num1* field refers to the maximum record size. The *num2* field refers to the working buffer size.

To achieve maximum performance, ensure that the working buffer size is large enough to hold any records that are written plus the control words (control words consist of 8 bytes per record). If a record plus control words are larger than the buffer, the layer must perform some inefficient operations to do the write. If the buffer is large enough, these operations can be avoided.

On reads, the buffer size is not as important, although larger sizes will usually perform better.

If the next lower-level layer is magnetic tape, this layer does not support I/O.

Table 13-9 Data manipulation: `f77` layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Record	Yes	`x` records

Table 13-10 Supported operations: `f77` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		Yes	
`ffreada`	Yes	Always synchronous	No	
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	
`ffwritea`	Yes	Always synchronous	No	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	
`ffweof`	Passed through		Yes	Only if explicitly requested
`ffweod`	Yes		Yes	

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffseek`	Yes	`ffseek(fd,0,0)` equals rewind; `ffseek(fd,0,2)` seeks to end	Yes	
`ffpos`	Yes		NA	
`ffbksp`	Yes	Only in lower-level layer	No	

The `fd` Layer

The `fd` layer allows connection of a FFIO file to a system file descriptor. You must specify the `fd` layer, as follows:

`fd:`[*num1*]

The keyword specification is as follows:

`fd`[`.file_descriptor=`*num1*]

The *num1* argument must be a system file descriptor for an open file. The `ffopen` or `ffopens` request opens a FFIO file descriptor that is connected to the specified file descriptor. The file connection does not affect the file whose name is passed to `ffopen`.

All other properties of this layer are the same as the `system` layer. See "The `system` Layer", page 106, for details.

The `global` Layer

The `global` layer is a caching layer that distributes data across all multiple SHMEM or MPI processes. Open and close operations require participation by all processes which access the file; all other operations are independently performed by one or more processes.

The following is the syntax for the `global` layer:

```
global[. type]:[num1]:[num2]
```

The following is the syntax for the keyword specification:

```
global[. type][.page_size=num1][.num_pages=num2]
```

The *type* argument can be `privpos` (default), in which is the file position is private to a process or `globpos` (deferred implementation), in which the file position is global to all processes.

The *num1* argument specifies the size in 4096–byte blocks of each cache page. *num2* specifies the number of cache pages to be used on each process. If there are *n* processes, then $n \times num2$ cache pages are used.

num2 buffer pages are allocated on every process which shares access to a global file. File pages are direct-mapped onto processes such that page *n* of the file will always be cached on process (*n* `mod NPES`), where `NPES` is the total number of processes sharing access to the global file. Once the process is identified where caching of the file page will occur, a least-recently-used method is used to assign the file page to a cache page within the caching process.

Table 13-11 Data manipulation: `global` layer

Granularity	Data model	Truncate on write
8 bits	Stream	No

Table 13-12 Supported operations: `global` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		No	

	Supported operations		Required of next lower level?	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffreada`	Yes	Always synchronous	Yes	
`ffreadc`	Yes		No	
`ffwrite`	Yes		No	
`ffwritea`	Yes	Always synchronous	Yes	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	
`ffweof`	No		No	
`ffweod`	Yes		Yes	
`ffseek`	Yes		Yes	Requires underlying interface to be a stream
`ffpos`	Yes		NA	
`ffbksp`	No		NA	

The `null` layer

The `null` layer is a syntactic convenience for users; it has no effect. This layer is commonly used to simplify the writing of a shell script when a shell variable is used to specify a FFIO layer specification. For example, the following is a line from a shell script with a tape file using the `assign` command and overlying blocking is expected on the tape (as specified by `BLKTYP`):

```
assign -F $BLKTYP,bmx fort.1
```

If `BLKTYP` is undefined, the illegal specification list `,bmx` results. The existence of the `null` layer lets the programmer set `BLKTYP` to `null` as a default, and simplify the script, as in the following:

```
assign -F null,bmx fort.1
```

This is identical to the following command:

```
assign -F bmx fort.1
```

The `syscall` Layer

The `syscall` layer directly maps each request to an appropriate system call. It has one optional parameter, as follows:

```
syscall[.cboption]
```

The *cboption* argument can have one of the following values:

`aiocb`	The `syscall` layer will be notified, via a signal, when the asynchronous I/O is completed.
`noaiocb`	The `syscall` layer will poll the completion status word to determine asynchronous I/O completion. This is the default value.

Table 13-13 Data manipulation: `syscall` layer

Granularity	Data model	Truncate on write
8 bits (1 byte)	Stream	No

Table 13-14 Supported operations: `syscall` layer

Operation	**Supported**	**Comments**
`ffopen`	Yes	`open`
`ffread`	Yes	`read`
`ffreada`	Yes	`reada(aio.read` on IRIX systems
`ffreadc`	Yes	`read` plus code
`ffwrite`	Yes	`write`

Operation	**Supported**	**Comments**
`ffwritea`	Yes	`writea` (`aio.write` on IRIX systems
`ffwritec`	Yes	`write` plus code
`ffclose`	Yes	`close`
`ffflush`	Yes	None
`ffweof`	No	None
`ffweod`	Yes	`trunc`(2)
`ffseek`	Yes	`lseek`(2)
`ffpos`	Yes	
`ffbksp`	No	

Lower-level layers are not allowed.

The `system` Layer

The `system` layer is implicitly appended to all specification lists, if not explicitly added by the user (unless the `syscall`, or `fd` layer is specified). It maps requests to appropriate system calls.

If the file that is opened is a tape file, the `system` layer becomes the `tape` layer.

For a description of options, see the `syscall` layer. Lower-level layers are not allowed.

The `text` Layer

The `text` layer performs text blocking by terminating each record with a newline character. It can also recognize and represent the EOF mark. The `text` layer is used with character files and does not work with binary data. The general specification follows:

```
text[.type]:[num1]:[num2]
```

The keyword specification follows:

```
text[.type][.newline=num1][.bufsize=num2]
```

The *type* field can have one of the following three values:

Value	**Definition**
`nl`	Newline-separated records.
`eof`	Newline-separated records with a special string such as `~e`. More than one EOF in a file is allowed.
`c205`	CYBER 205–style text file (on the CYBER 205, these are called R-type records).

The *num1* field is the decimal value of a single character that represents the newline character. The default value is 10 (octal 012, ASCII line feed).

The *num2* field specifies the working buffer size (in decimal bytes). If any lower-level layers are record oriented, this is also the block size.

Table 13-15 Data manipulation: `text` layer

Granularity	Data model	Truncate on write
8 bits	Record.	No

Table 13-16 Supported operations: `text` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		Yes	
`ffreada`	Yes	Always synchronous	No	

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	
`ffwritea`	Yes	Always synchronous	No	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	
`ffweof`	Passed through		Yes	Only if explicitly requested
`ffweod`	Yes		Yes	
`ffseek`	Yes		Yes	
`ffpos`	Yes		No	
`ffbksp`	No		No	

The `user` and `site` Layers

The `user` and `site` layers let users and site administrators build layers that meet specific needs. The syntax follows:

```
user[num1]:[num2]
```

```
site:[num1]:[num2]
```

The open processing passes the *num1* and *num2* arguments to the layer and are interpreted by the layers.

See “Creating a `user` Layer,” Chapter 14, "Creating a `user` Layer ", page 113 for an example of how to create an FFIO layer.

The `vms` Layer

The `vms` layer handles record blocking for three common record types on VAX/VMS operating systems. The general format of the specification follows.

```
vms.[type.subtype]:[num1]:[num2]
```

The following is the alternate keyword specification for this layer:

```
vms.[type.subtype][.recsize=num1][.mbs=num2]
```

The following *type* values are supported:

Value	**Definition**
`f`	VAX/VMS fixed-length records
`v`	VAX/VMS variable-length records
`s`	VAX/VMS variable-length segmented records

In addition to the record type, you must specify a record subtype, which has one of the following four values:

Value	**Definition**
`bb`	Format used for binary blocked transfers
`disk`	Same as binary blocked
`tr`	Transparent format, for files transferred as a bit stream to and from the VAX/VMS system
`tape`	VAX/VMS labeled tape

The *num1* field is the maximum record size that may be read or written. It is ignored by the `s` record type.

Table 13-17 Values for record size: `vms` layer

Field	Minimum	Maximum	Default	Comments
`v.bb`	1	32,767	32,767	
`v.tape`	1	9995	2043	
`v.tr`	1	32,767	2044	
`s.bb`	1	None	None	No maximum record size
`s.tape`	1	None	None	No maximum record size
`s.tr`	1	None	None	No maximum record size

The *num2* field is the maximum segment or block size that is allowed on input and is produced on output. For `vms.f.tr` and `vms.f.bb`, *num2* should be equal to the record size (*num1*). Because `vms.f.tape` places one or more records in each block, `vms.f.tape` *num2* must be greater than or equal to *num1*.

Table 13-18 Values for maximum block size: `vms` layer

Field	Minimum	Maximum	Default	Comments
`v.bb`	1	32,767	32,767	
`v.tape`	6	32,767	2,048	
`v.tr`	3	32,767	32,767	N/A
`s.bb`	5	32,767	2,046	
`s.tape`	7	32,767	2,048	
`s.tr`	5	32,767	2,046	N/A

For `vms.v.bb` and `vms.v.disk` records, *num2* is a limit on the maximum record size. For `vms.v.tape` records, it is the maximum size of a block on tape; more specifically, it is the maximum size of a record that will be written to the next lower layer. If that layer is `tape`, *num2* is the tape block size. If it is `cos`, it will be a COS record that represents a tape block. One or more records are placed in each block.

For segmented records, *num2* is a limit on the **block** size that will be produced. No limit on record size exists. For `vms.s.tr` and `vms.s.bb`, the block size is an upper limit on the size of a segment. For `vms.s.tape`, one or more segments are placed in a tape block. It functions as an upper limit on the size of a segment and a preferred tape block size.

Table 13-19 Data manipulation: `vms` layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Record	No for `f` records. Yes for `v` and `s` records.	`f` records for `f` formats. `v` records for `v` formats.

Table 13-20 Supported operations: `vms` layer

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	
`ffread`	Yes		Yes	
`ffreada`	Yes	Always synchronous	No	
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	
`ffwritea`	Yes	Always synchronous	No	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes		No	
`ffweof`	Yes and passed through	Yes for `s` records; passed through for others	Yes	Only if explicitly requested
`ffweod`	Yes		Yes	

	Supported operations		**Required of next lower level?**	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffseek`	Yes	`seek(fd,0,0)` only (equals rewind)	Yes	`seek(fd,0,0)` only
`ffpos`	Yes		NA	
`ffbksp`	No		No	

Creating a `user` Layer

This chapter explains some of the internals of the FFIO system and explains the ways in which you can put together a `user` or `site` layer. "`user` Layer Example", page 116, is an example of a `user` layer.

Internal Functions

The FFIO system has an internal model of data that maps to any given actual logical file type based on the following concepts:

- Data is a stream of bits. Layers must declare their granularity by using the `fffcntl`(3c) call.
- Record marks are boundaries between logical records.
- End-of-file marks (EOF) are a special type of record that exists in some file structures.
- End-of-data (EOD) is a point immediately beyond the last data bit, EOR, or EOF in the file. You cannot read past or write after an EOD. In a case when a file is positioned after an EOD, a write operation (if valid) immediately moves the EOD to a point after the last data bit, end-of-record (EOR), or EOF produced by the write.

All files are streams that contain zero or more data bits that may contain record or file marks.

No inherent hierarchy or ordering is imposed on the file structures. Any number of data bits or EOR and EOF marks may appear in any order. The EOD, if present, is by definition last. Given the EOR, EOF, and EOD return statuses from read operations, only EOR may be returned along with data. When data bits are immediately followed by EOF, the record is terminated implicitly.

Individual layers can impose restrictions for specific file structures that are more restrictive than the preceding rules. For instance, in COS blocked files, an EOR always immediately precedes an EOF.

Successful mappings were used for all logical file types supported, except formats that have more than one type of partitioning for files (such as end-of-group or more than one level of EOF). For example, some CDC file formats have level numbers in the partitions. FFIO and CDC map level 017 to an EOF. No other handling is provided for these level numbers.

Internally, there are two main protocol components: the operations and the stat structure.

The Operations Structure

Many of the operations try to mimic system calls. In the man pages for `ffread`(3c), `ffwrite`(3c), and others, the calls can be made without the optional parameters and appear like the system calls. Internally, all parameters are required.

The following list is a brief synopsis of the interface routines that are supported at the user level. Each of these `ff` entry points checks the parameters and issues the corresponding internal call. Each interface routine provides defaults and dummy arguments for those optional arguments that the user does not provide.

Each layer must have an internal entry point for all of these operations; although in some cases, the entry point may simply issue an error or do nothing. For example, the `syscall` layer uses `_ff_noop` for the `ffflush` entry point because it has no buffer to flush, and it uses `_ff_err2` for the `ffweof` entry point because it has no representation for EOF. No optional parameters for calls to the internal entry points exist. All arguments are required.

A list of operations called as functions from a C program follows:

```
fd = ffopen(file, flags, mode, stat);
nb = ffread(fd, buf, nb, stat, fulp, &ubc);
opos = ffseek(fd, pos, whence, stat);
nb = ffreada(fd, buf, nb, stat, fulp, &ubc);
ret = ffpos(fd,cmd, argp, len, stat)
ret = fffcntl(fd, cmd, arg, stat);
nb = ffwritea(fd, buf, nb, stat, fulp, &ubc);
```

The following are the variables for the internal entry points and the variable definitions. An internal entry point must be provided for all of these operations:

Variable	**Definition**
fd	The FFIO pointer `(struct fdinfo *)fd`.
file	A `char*` file.

flags	File status flag for open, such as `O_RDONLY`.
buf	Bit pointer to the user data.
nb	Number of bytes.
ret	The status returned; >=0 is valid, <0 is error.
stat	A pointer to the status structure.
fulp	The value `FULL` or `PARTIAL` defined in `ffio.h` for full or partial-record mode.
&ubc	A pointer to the unused bit count; this ranges from 0 to 7 and represents the bits not used in the last byte of the operation. It is used for both input and output.
pos	A byte position in the file.
`opos`	The old position of the file, just like the `system` call.
whence	The same as the `syscall`.
cmd	The command request to the `fffcntl`(3c) call.
arg	A generic pointer to the `fffcntl` argument.
mode	Bit pattern denoting file's access permissions.
argp	A pointer to the input or output data.
len	The length of the space available at *argp*. It is used primarily on output to avoid overwriting the available memory.

FFIO and the Stat Structure

The *stat* structure contains four fields in the current implementation. They mimic the *iosw* structure of the `ASYNC` syscalls to the extent possible. All operations are expected to update the *stat* structure on each call. The `SETSTAT` and `ERETURN` macros are provided in `ffio.h` for this purpose.

The fields in the *stat* structure are as follows:

Status field	**Description**
`stat.sw_flag`	0 indicates outstanding; 1 indicates I/O complete.
`stat.sw_error`	0 indicates no error; otherwise, the error number.

`stat.sw_count`	Number of bytes transferred in this request. This number is rounded up to the next integral value if a partial byte is transferred.
`stat.sw_stat`	This tells the status of the I/O operation. The `FFSTAT(stat)` macro accesses this field. The following are the possible values: `FFBOD`: At beginning-of-data (BOD). `FFCNT`: Request terminated by count (either the count of bytes before EOF or EOD in the file or the count of the request). `FFEOR`: Request termination by EOR or a full record mode read was processed. `FFEOF`: EOF encountered. `FFEOD`: EOD encountered. `FFERR`: Error encountered.

If `count` is satisfied simultaneously with EOR, the `FFEOR` is returned.

The EOF and EOD status values must never be returned with data. This means that if a byte-stream file is being traversed and the file contains 100 bytes and then an EOD, a read of 500 bytes will return with a `stat` value of `FFCNT` and a return byte count of 100. The next read operation returns `FFEOD` and a `count` of 0.

A `FFEOF` or `FFEOD` status is always returned with a zero-byte transfer count.

`user` Layer Example

This section gives a complete and working `user` layer. It traces I/O at a given level. All operations are passed through to the next lower-level layer, and a `trace` record is sent to the `trace` file.

The first step in generating a user layer is to create a table that contains the addresses for the routines which fulfill the required functions described in "The Operations Structure", page 114, and "FFIO and the Stat Structure", page 115. The format of the table can be found in `struct xtr_s`, which is found in the `<ffio.h>` file. No restriction is placed on the names of the routines, but the table must be called `_usr_ffvect` for it to be recognized as a `user` layer. In the example, the declaration of the table can be found with the code in the `_usr_open` routine.

To use this layer, you must take advantage of the `soft` external files in the library.

On IRIX systems, to build for the `n32` ABI on MIPS3 architectures:

```
cc -c  -n32 -mips3 usr*.c -D_LIB_INTERNAL
f90 -n32 -mips3 usr*.o main.f -o abs
assign -F user,others... fort.1
./abs
```

```
static char USMID[] = "@(#)code/usrbksp.c       1.0    ";
/*   COPYRIGHT SGI
*    UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
*    THE COPYRIGHT LAWS OF THE UNITED STATES.
*/
#include <ffio.h>
#include "usrio.h"
/*
*    trace backspace requests
*/
int
_usr_bksp(struct fdinfo *fio, struct ffsw *stat)
     {
     struct fdinfo *llfio;
     int ret;

     llfio = fio->fioptr;
     _usr_enter(fio, TRC_BKSP);
     _usr_pr_2p(fio, stat);
     ret = XRCALL(llfio, backrtn) llfio, stat);
     _usr_exit(fio, ret, stat);
    return(0);
    }
```

```
 static char USMID[] = "@(#)code.usrclose.c     1.0    ";
/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <malloc.h>
#include <ffio.h>
#include "usrio.h"
/*
 *  trace close requests
 */
int
_usr_close(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        struct trace_f *pinfo;
        int ret;
        llfio = fio->fioptr;
/*
 *  lyr_info is a place in the fdinfo block that holds
 *  a pointer to the layer's private information.
 */
        pinfo = (struct trace_f *)fio->lyr_info;

        _usr_enter(fio, TRC_CLOSE);
        _usr_pr_2p(fio, stat);
/*
 *  close file
 */
        ret = XRCALL(llfio, closertn) llfio, stat);
/*
 *  It is the layer's responsibility to clean up its mess.
 */
        free(pinfo->name);
        pinfo->name = NULL;
        free(pinfo);
        _usr_exit(fio, ret, stat);
        (void) close(pinfo->usrfd);
        return(0);
        }
```

```
static char USMID[] = "@(#)code/usrfcntl.c       1.0    ";
/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <ffio.h>
#include "usrio.h"
/*
 *  trace fcntl requests
 *
 *  Parameters:
 *   fd      - fdinfo pointer
 *   cmd     - command code
 *   arg     - command specific parameter
 *   stat    - pointer to status return word
 *
 * This fcntl routine passes the request down to the next lower
 * layer, so it provides nothing of its own.
 *
 * When writing a user layer, the fcntl routine must be provided,
 * and must provide correct responses to one essential function and
 * two desirable functions.
 *
 *  FC_GETINFO: (essential)
 *  If the 'cmd' argument is FC_GETINFO, the fields of the 'arg' is
 *  considered a pointer to an ffc_info_s structure, and the fields
 *  must be filled. The most important of these is the ffc_flags
 *  field, whose bits are defined in <ffio.h>.(Look for FFC_STRM
 *  through FFC_NOTRN)
 *  FC_STAT: (desirable)
 *  FC_RECALL: (desirable)
 */
int
_usr_fcntl(struct fdinfo *fio, int cmd, void *arg, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        struct trace_f *pinfo;
        int ret;

        llfio = fio->fioptr;
        pinfo = (struct trace_f *)fio->lyr_info;
        _usr_enter(fio, TRC_FCNTL);
        _usr_info(fio, "cmd=%d ", cmd);
        ret = XRCALL(llfio, fcntlrtn) llfio, cmd, arg, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }
```

```
static char USMID[] = "@(#)code/usropen.c       1.0    ";

/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <fcntl.h>
#include <malloc.h>
#include <ffio.h>
#include "usrio.h"
#define SUFFIX      ".trc"

/*
 * trace open requests;
 * The following routines compose the user layer. They are declared
 * in "usrio.h"
 */

/*
 * Create the _usr_ffvect structure.  Note the _ff_err inclusion to
 * account for the listiortn, which is not supported by this user
 * layer
 */
struct xtr_s _usr_ffvect =
  {
  _usr_open,   _usr_read,   _usr_reada,   _usr_readc,
  _usr_write,  _usr_writea, _usr_writec,  _usr_close,
  _usr_flush,  _usr_weof,   _usr_weod,    _usr_seek,
  _usr_bksp,   _usr_pos,    _usr_err,     _usr_fcntl
   };

_ffopen_t
_usr_open(
         const char *name,
         int flags,
         mode_t mode,
         struct fdinfo * fio,
         union spec_u *spec,
         struct ffsw *stat,
         long cbits,
         int cblks,
         struct gl_o_inf *oinf)
         {
         union spec_u *nspec;
         struct fdinfo *llfio;
         struct trace_f *pinfo;
         char *ptr = NULL;
         int  namlen, usrfd;
         _ffopen_t nextfio;
         char buf[256];

         namlen = strlen(name);
         ptr = malloc(namlen + strlen(SUFFIX) + 1);
         if (ptr == NULL) goto badopen;
         pinfo = (struct trace_f *)malloc(sizeof(struct trace_f));
         if (pinfo == NULL) goto badopen;

         fio->lyr_info = (char *)pinfo;
/*
 *  Now, build the name of the trace info file, and open it.
 */
         strcpy(ptr, name);
         strcat(ptr, SUFFIX);
         usrfd = open(ptr, O_WRONLY | O_APPEND | O_CREAT, 0666);
/*
 *  Put the file info into the private data area.
 */
         pinfo->name = ptr;
         pinfo->usrfd = usrfd;
         ptr[namlen] = ’\0’;
/*
 *  Log the open call
 */
         _usr_enter(fio, TRC_OPEN);
         sprintf(buf,"(\"%s\", %o, %o...);\n", name, flags, mode);
         _usr_info(fio, buf, 0);
/*
 *  Now, open the lower layers
 */
         nspec = spec;
         NEXT_SPEC(nspec);
         nextfio = _ffopen(name, flags, mode, nspec, stat, cbits, cblks,
                    NULL, oinf);
         _usr_exit_ff(fio, nextfio, stat);
         if (nextfio != _FFOPEN_ERR)
                {
                DUMP_IOB(fio); /* debugging only */
                return(nextfio);
                }
/*
 *  End up here only on an error
 *
 */

badopen:
        if(ptr != NULL) free(ptr);
        if (fio->lyr_info != NULL) free(fio->lyr_info);
        _SETERROR(stat, FDC_ERR_NOMEM, 0);
         return(_FFOPEN_ERR);
        }
        _usr_err(struct fdinfo *fio)
        {
         _usr_info(fio,"ERROR: not expecting this routine\n",0);
         return(0);
        }
```

```
static char USMID[] = "@(#)code/usrpos.c       1.1    ";

/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#include <ffio.h>
#include "usrio.h"

/*
 *  trace positioning requests
 */

_ffseek_t
_usr_pos(struct fdinfo *fio, int cmd, void *arg, int len, struct ffsw *stat)
   {
   struct fdinfo *llfio;
   struct trace_f *usr_info;
   _ffseek_t ret;

   llfio = fio->fioptr;
   usr_info = (struct trace_f *)fio->lyr_info;

   _usr_enter(fio,TRC_POS);
   _usr_info(fio, " ", 0);
   ret = XRCALL(llfio, posrtn) llfio, cmd, arg, len, stat);
   _usr_exit_sk(fio, ret, stat);
   return(ret);
   }
```

```
static char USMID[] = "@(#)code/usrprint.c       1.1    ";

/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <ffio.h>
#include "usrio.h"

static char *name_tab[] =
                        {
                        "???",
                        "ffopen",
                        "ffread",
                        "ffreada",
                        "ffreadc",
                        "ffwrite",
                        "ffwritea",
                        "ffwritec",
                        "ffclose",
                        "ffflush",
                        "ffweof",
                        "ffweod",
                        "ffseek",
                        "ffbksp",
                        "ffpos",
                        "fflistio",
                        "fffcntl",
                        };

/*
 * trace printing stuff
 */
int
_usr_enter(struct fdinfo *fio, int opcd)
                {
                char buf[256], *op;
                struct trace_f *usr_info;
                op = name_tab[opcd];
                usr_info = (struct trace_f *)fio->lyr_info;
                sprintf(buf, "TRCE: %s ",op);
                write(usr_info->usrfd, buf, strlen(buf));
                return(0);
                }

void
_usr_info(struct fdinfo *fio, char *str, int arg1)
       {
       char buf[256];
       struct trace_f *usr_info;

       usr_info = (struct trace_f *)fio->lyr_info;
       sprintf(buf, str, arg1);
       write(usr_info->usrfd, buf, strlen(buf));
       }

void
_usr_exit(struct fdinfo *fio, int ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
        write(usr_info->usrfd, buf, strlen(buf));
        }

void
_usr_exit_ss(struct fdinfo *fio, ssize_t ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
```

```
#ifdef __mips
#if (_MIPS_SZLONG== 32)
        sprintf(buf, "TRCX:   ret=%lld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#else
        sprintf(buf, "TRCX:   ret=%ld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
#else
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
        }

void
_usr_exit_ff(struct fdinfo *fio, _ffopen_t ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
#ifdef __mips
        sprintf(buf, "TRCX:   ret=%lx, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#else
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
 }
void
_usr_exit_sk(struct fdinfo *fio, _ffseek_t ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;
        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
#ifdef __mips
#if (_MIPS_SZLONG== 32)
```

```
        sprintf(buf, "TRCX:   ret=%lld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#else
        sprintf(buf, "TRCX:   ret=%ld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
#else
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_rwc(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
#ifdef __mips
#if (_MIPS_SZLONG == 64) && (_MIPS_SZPTR == 64)
        sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
#else if (_MIPS_SZLONG == 32) && (_MIPS_SZPTR == 32)
        sprintf(buf,"(fd / %lx */, &memc[%lx], %lld, &statw[%lx], ",
            fio, BPTR2CP(bufptr), nbytes, stat);
#endif
#else
        sprintf(buf,"(fd / %x */, &memc[%x], %d, &statw[%x], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
        if (fulp == FULL)
              sprintf(buf,"FULL");
       else
               sprintf(buf,"PARTIAL");
              write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_rww(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
#ifdef __mips
#if (_MIPS_SZLONG == 64) && (_MIPS_SZPTR == 64)
        sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
#else if (_MIPS_SZLONG == 32) && (_MIPS_SZPTR == 32)
        sprintf(buf,"(fd / %lx */, &memc[%lx], %lld, &statw[%lx], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
#endif
#else
        sprintf(buf,"(fd / %x */, &memc[%x], %d, &statw[%x], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
        if (fulp == FULL)
             sprintf(buf,"FULL");
        else
        sprintf(buf,"PARTIAL");
        write(usr_info->usrfd, buf, strlen(buf));
        sprintf(buf,", &conubc[%d]; ", *ubc);
        write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_2p(struct fdinfo *fio, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;
        usr_info = (struct trace_f *)fio->lyr_info;
#ifdef __mips
#if (_MIPS_SZLONG == 64) && (_MIPS_SZPTR == 64)
        sprintf(buf,"(fd / %lx */,  &statw[%lx], ",
             fio,  stat);
#else if (_MIPS_SZLONG == 32) && (_MIPS_SZPTR == 32)
         sprintf(buf,"(fd / %lx */, &statw[%lx], ",
              fio,  stat);
#endif
#else
         sprintf(buf,"(fd / %x */, &statw[%x], ",
              fio,  stat);
#endif
         write(usr_info->usrfd, buf, strlen(buf));
         }
```

```
 static char USMID[] = "@(#)code/usrread.c       1.0    ";
/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#include <ffio.h>
#include "usrio.h"

/*
 * trace read requests
 *
 * Parameters:
 *  fio     - Pointer to fdinfo block
 *  bufptr  - bit pointer to where data is to go.
 *  nbytes  - Number of bytes to be read
 *  stat    - pointer to status return word
 *  fulp    - full or partial read mode flag
 *  ubc     - pointer to unused bit count
 */
ssize_t
_usr_read(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
     {
     struct fdinfo *llfio;
     char *str;
     ssize_t ret;
     llfio = fio->fioptr;
     _usr_enter(fio, TRC_READ);
     _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
     ret = XRCALL(llfio, readrtn) llfio, bufptr, nbytes, stat,
          fulp, ubc);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
     }
```

```
/*
 * trace reada (asynchronous read) requests
 *
 * Parameters:
 *  fio     - Pointer to fdinfo block
 *  bufptr  - bit pointer to where data is to go.
 *  nbytes  - Number of bytes to be read
 *  stat    - pointer to status return word
 *  fulp    - full or partial read mode flag
 *  ubc     - pointer to unused bit count
 */
ssize_t
_usr_reada(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
    {
    struct fdinfo *llfio;
    char *str;
    ssize_t ret;

    llfio = fio->fioptr;
    _usr_enter(fio, TRC_READA);
    _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
    ret = XRCALL(llfio,readartn)llfio,bufptr,nbytes,stat,fulp,ubc);
    _usr_exit_ss(fio, ret, stat);
    return(ret);
    }

/*
* trace readc requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be read
*  stat    - pointer to status return word
*  fulp    - full or partial read mode flag
*/
ssize_t
_usr_readc(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
     {
     struct fdinfo *llfio;
     char *str;
     ssize_t ret;
     llfio = fio->fioptr;
     _usr_enter(fio, TRC_READC);
     _usr_pr_rwc(fio, bufptr, nbytes, stat, fulp);
     ret = XRCALL(llfio, readcrtn)llfio, bufptr, nbytes, stat,
         fulp);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
        }

/*
* _usr_seek()
*
* The user seek call should mimic the lseek system call as
* much as possible.
*/
_ffseek_t
_usr_seek(
struct fdinfo *fio,
off_t pos,
int whence,
struct ffsw *stat)
        {
        struct fdinfo *llfio;
        _ffseek_t ret;
        char buf[256];

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_SEEK);
```

```
#ifdef __mips
#if (_MIPS_SZLONG == 64)
        sprintf(buf,"pos %ld, whence %d\n", pos, whence);
#else
        sprintf(buf,"pos %lld, whence %d\n", pos, whence);
#endif
#else
        sprintf(buf,"pos %d, whence %d\n", pos, whence);
#endif
        _usr_info(fio, buf, 0);
        ret = XRCALL(llfio, seekrtn) llfio, pos, whence, stat);
        _usr_exit_sk(fio, ret, stat);
        return(ret);
        }
```

```
static char USMID[] = "@(#)code/usrwrite.c       1.0    ";

/*  COPYRIGHT SGI
*  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
*  THE COPYRIGHT LAWS OF THE UNITED STATES.
*/

#include <ffio.h>
#include "usrio.h"

/*
* trace write requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*  ubc     - pointer to unused bit count (not used for IBM)
*/
ssize_t
_usr_write(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
     {
     struct fdinfo *llfio;
     ssize_t ret;

     llfio = fio->fioptr;
     _usr_enter(fio, TRC_WRITE);
     _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
     ret = XRCALL(llfio, writertn) llfio, bufptr, nbytes, stat,
          fulp,ubc);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
     }

/*
* trace writea requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*  ubc     - pointer to unused bit count (not used for IBM)
*/
ssize_t
_usr_writea(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
    {
    struct fdinfo *llfio;
    ssize_t ret;

    llfio = fio->fioptr;
    _usr_enter(fio, TRC_WRITEA);
    _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
    ret = XRCALL(llfio, writeartn) llfio, bufptr, nbytes, stat,
        fulp,ubc);
    _usr_exit_ss(fio, ret, stat);
    return(ret);
    }

/*
* trace writec requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*/

ssize_t
_usr_writec(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
        {
        struct fdinfo *llfio;
        ssize_t ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WRITEC);
        _usr_pr_rwc(fio, bufptr, nbytes, stat, fulp);
        ret = XRCALL(llfio, writecrtn)llfio,bufptr, nbytes, stat,
           fulp);
        _usr_exit_ss(fio, ret, stat);
        return(ret);
        }
/*
* Flush the buffer and clean up
* This routine should return 0, or -1 on error.
*/
int
_usr_flush(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;
        llfio = fio->fioptr;

        _usr_enter(fio, TRC_FLUSH);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, flushrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }
```

```
/*
* trace WEOF calls
*
* The EOF is a very specific concept.   Don’t confuse it with the
* EOF, or the trunc(2) system call.
*/
int
_usr_weof(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WEOF);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, weofrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }

/*
* trace WEOD calls
*
* The EOD is a specific concept.  Don’t confuse it with the
*  EOF.  It is usually mapped to the trunc(2) system call.
*/
int
_usr_weod(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WEOD);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, weodrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }
```

```
/* USMID @(#)code/usrio.h       1.1    */

/*  COPYRIGHT SGI
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#define TRC_OPEN 1
#define TRC_READ 2
#define TRC_READA 3
#define TRC_READC 4
#define TRC_WRITE 5
#define TRC_WRITEA 6
#define TRC_WRITEC 7
#define TRC_CLOSE 8
#define TRC_FLUSH 9
#define TRC_WEOF 10
#define TRC_WEOD 11
#define TRC_SEEK 12
#define TRC_BKSP 13
#define TRC_POS 14
#define TRC_UNUSED 15
#define TRC_FCNTL 16

struct trace_f
        {
        char    *name;          /* name of the file */
        int   usrfd;            /* file descriptor of trace file */
        };
/*
 * Prototypes
 */
extern int _usr_bksp(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_close(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_fcntl(struct fdinfo *fio, int cmd, void *arg,
           struct ffsw *stat);
extern _ffopen_t _usr_open(const char *name, int flags,
           mode_t mode, struct fdinfo * fio, union spec_u *spec,
           struct ffsw *stat, long cbits, int cblks,
           struct gl_o_inf *oinf);
extern int _usr_flush(struct fdinfo *fio, struct ffsw *stat);
extern _ffseek_t _usr_pos(struct fdinfo *fio, int cmd, void *arg,
           int len, struct ffsw *stat);
extern ssize_t _usr_read(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_reada(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_readc(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp);
extern _ffseek_t _usr_seek(struct fdinfo *fio, off_t pos, int whence,
           struct ffsw *stat);
extern ssize_t _usr_write(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_writea(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_writec(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp);
extern int _usr_weod(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_weof(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_err();

/*
 * Prototypes for routines that are used by the user layer.
 */
extern int _usr_enter(struct fdinfo *fio, int opcd);
extern void _usr_info(struct fdinfo *fio, char *str, int arg1);
extern void _usr_exit(struct fdinfo *fio, int ret, struct ffsw *stat);
extern void _usr_exit_ss(struct fdinfo *fio, ssize_t ret,
           struct ffsw *stat);
extern void _usr_exit_ff(struct fdinfo *fio, _ffopen_t ret,
           struct ffsw *stat);
extern void _usr_exit_sk(struct fdinfo *fio, _ffseek_t ret,
           struct ffsw *stat);
extern void _usr_pr_rww(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern void _usr_pr_2p(struct fdinfo *fio, struct ffsw *stat);
```

Glossary

blocking

In parallel processing, a blocking function is one that does not return until the function is complete.

disk striping

(1) Multiplexing or interleaving a disk file across two or more disk drives to enhance I/O performance. The performance gain is function of the number of drives and channels used.

file system

(1) The disks located in the fileserver that contain directories. (2) An individual partition or cluster that has been formatted properly. The root file system is always mounted; other file systems are mounted as needed. (3) The entire set of available disk space. (4) A structure used to store programs and files on disk. A file system can be mounted (accessible for operations) or unmounted (noninteractive and unavailable for system use).

logical device

One or more physical device slices that the operating system treats as a single device.

raw I/O

A method of performing input/output in UNIX in which the programmer must handle all of the I/O control. This is basically unformatted I/O. The opposite of "raw I/O" is "cooked I/O" (UNIX humor).

record

(1) A group of contiguous words or characters that are related by convention. A record may be fixed or of variable length. (2) A record for a listable data set; each line is a record. (3) Each module of a binary-load data set is a record.

slice

(1) As used in the context of the low-speed communication (networking) subsystem in an EIOP, a slice is a subdivision of a channel buffer; sections of the buffer are divided into slices used for buffering network messages and data.

stream

(1) A software path of messages related to one file. (2) A stream, or logical command queue, is associated with a slave in the intelligent peripheral interface (IPI) context. The stream is used in identifying IPI-3 commands destined for that slave. A slave may have 0, 1, or many streams associated with it at any given time.

unit

When used in the context of disk software on the IOS-E, unit refers to one disk drive that is daisy-chained with others on one channel adapter. The unit number represents an ordinal for referring to one disk on the channel.

Index

A

assign
- and Fortran I/O, 36
  - alternative file names, 36
  - buffer size selection, 39
  - direct-access I/O tuning, 40
  - file structure selection, 38
  - foreign file format specification, 40
  - Fortran file truncation, 40
- assign basics, 31
  - assign command, 32
  - open processing, 31
  - related library routines, 35
- local assign, 42

assign command
- syntax, 32

assign environment, 31
- related library routines, 35

assign library routines
- calling sequences, 35

auxiliary I/O, 16

B

bin processing, 46

blocked file structure, 47

bufa layer, 65, 92

BUFFER IN/BUFFER OUT, 20
- advantages, 19

buffer size considerations, 64

buffer size specification, 39

buffering, 51
- introduction to, 51
- library buffering, 53
- overview, 51
- system cache, 53
- unbuffered I/O, 53

buffers
- usage, 51

C

C I/O
- C I/O from Fortran, 28
- FILE type
  - usage, 28
- functions, 28
- mixing Fortran and C I/O, 28

cache layer, 65, 93
- and improved I/O performance, 65
- specification, 65

cachea layer, 65, 95

characteristics of individual layers, 89
- data model, 90
- granularity, 90
- implementation strategy, 90
- truncate on write, 90

conversion methods
- advantages and disadvantages, 73

cos file structure, 47

COS blocked file structure
- and ENDFILE records, 47
- example
  - formatted file, 39

COS blocked files
- and FFIO, 63

cos blocking layer, 97

creating an I/O layer, 113
- internal functions, 113
- operations structure, 114
- stat structure, 115

D

data conversion, 84
data copying, 85
data item conversion
  explicit conversion
    advantages/disadvantages, 73
  implicit conversion
    advantages/disadvantages, 74
data manipulation
  characteristics, 89
data output flow, 78
data transfer
  input statement
    READ, 11
  output statement
    PRINT, 11
    WRITE, 11
definitions
  external file, 5
  external unit identifier, 5
  file position, 8
  internal file, 5
  internal unit identifier, 5
direct access
  external file properties, 7
direct-access I/O tuning, 40
distributed I/O, 102

E

environment variables
  LISTIO_PRECISION, 15
EOF records
  in standard Fortran, 8
event layer, 99
examples
  assign -a, 36
  BACKSPACE statement, 18
  buffer size specification, 39
  COS blocked file structure
    formatted file, 39
  direct access edit-directed I/O statement, 12
  direct access unformatted I/O statement, 16
  ENDFILE statement, 18
  explicit named open statement, 10
  explicit unnamed open statement, 10
  file structure selection, 38
  Fortran interfaces to C functions, 29
  implicit open statement, 9
  ISHELL call, 21
  layered I/O, 59
  LENGTH function, 20
  list-directed READ statement, 15
  list-directed WRITE statement, 15
  local assign mode, 42
  named pipe, 21
  named pipes file structure, 22
  namelist I/O, 15
  OPEN statement, 16
  piped I/O with EOF detection, 24
  piped I/O with no EOF detection, 22
  sequential access edit-directed READ statement, 12
  sequential access edit-directed WRITE statement, 12
  sequential access unformatted READ statement, 16
  sequential access unformatted WRITE statement, 16
  specifying I/O class, 56
  specifying I/O processing steps, 57
    READ requests, 58
  unblocked file structure, 38
  UNIT function, 20
  user layer, 116
explicit data conversion
  definition, 67
explicit data item conversion, 68
explicit named open statement
  example, 10
explicit unnamed open statement

example, 10
external file, 5
external files
direct access, 7
format, 6
sequential access, 6
external unit identifier, 5
external units
and file connections, 6

F

f77 layer, 100
fd layer, 102
fdcp tool, 67
FFIO
and buffer size considerations, 64
and Fortran I/O forms, 57
and performance enhancements, 64
and reading and writing COS files, 63
and reading and writing fixed-length records, 63
and reading and writing unblocked files, 63
common formats, 62
error messages, 2
introduction, 55
reading and writing text files, 62
removing blocking, 64
using the bufa layer, 65
using the cache layer, 65
using the cachea layer, 65
FFIO and foreign data
foreign conversion tips
workstation and IEEE conversion, 74
FFIO and the stat structure, 115
FFIO layer reference
individual layers
bufa layer, 92
cache layer, 93
cachea layer, 95
COS blocking layer, 97
event layer, 99
f77 layer, 100
fd layer, 102
global layer, 102
null layer, 104
syscall layer, 105
system layer, 106
text layer, 106
user and site layers, 108
vms layer, 109
FFIO specifications
text files, 62
using with text files, 62
using with unblocked files, 63
file access, 6
direct access, 7
sequential access, 6
file connections
alternative file names, 36
tuning, 36
file positioning statement, 17
file properties, 6
file structure, 43
alternatives
using assign, 38
assign options, 43
COS file structure, 47
default, 38
selection, 38
text file structures, 46
unblocked file structure, 44
bin file processing, 46
sbin file processing, 45
u file processing, 46
file structure overhead, 82
file truncation
activating and suppressing, 40
FILE type
available buffering, 28
used with C I/O functions, 28
fixed-length records
and FFIO, 63

foreign conversion tips
  workstation and IEEE conversion, 74
foreign file conversion
  and fdcp, 67
  choosing conversion methods, 73
  conversion techniques, 74
  data item conversion, 68
  explicit data item conversion, 68
  implicit data item conversion, 69
  overview, 67
  routines, 68
foreign file format specifications, 40
foreign I/O formats
  supported data types, , 71
formatted I/O statements
  optimizing, 12
  types, 11
formatted record size, 85
Fortran I/O extensions, 19
  BUFFER IN/BUFFER OUT, 19
    LENGTH function, 20
    UNIT intrinsic routine, 20
Fortran standard
  auxiliary I/O statements
    BACKSPACE file positioning statement, 18
    ENDFILE file positioning statement, 18
    file connection statements, 16, 17
    file positioning statements, 17
    INQUIRE statement, 17
    OPEN, 16
    REWIND file positioning statement, 18
  auxilliary I/O statements, 16
  data transfer
    formatted I/O, 11
  data transfer statements, 11
    edit-directed formatted I/O, 12
    list-directed formatted I/O, 14
    namelist I/O, 15
    unformatted I/O, 16
  external files, 6
    file access, 6
  file name specification, 5
  file properties, 6
  file types, 5
  files
    direct file access, 7
    external files, 6
    file position, 8
    form, 6
    internal files, 5
    sequential file access, 6
  formatted I/O statements
    optimizing, 12
  Fortran unit identifiers, 8
  overview, 5
  overview of files, 5
Fortran unit identifiers, 8
  valid unit numbers, 9

G

global I/O, 102
global layer, 102

I

I/O forms
  and FFIO usage, 57
I/O layers, 58, 78
  supported operations, 91
  unblocked data transfer, 64
I/O optimization, 77
  avoiding formatted I/O, 84
  bypassing library buffers, 85
  characterizing files, 78
  data conversions, 84
  file structure overhead, 82
  library buffer sizes, 85
  optimizing speed, 80
  overlapping CPU and I/O, 86
  overview, 77

overview of optimization techniques, 79
source code changes, 80
summary of techniques, 79
system requests, 81
using asynchronous
read-ahead/write-behind, 83
using pipes, 86
using scratch files, 82
using simpler file structures, 84
using the cache layer, 81
I/O processing steps, 56
description, 55
I/O classes, 58
specifying I/O class, 56
example, 56
implicit data conversion
definition, 67
implicit data item conversion, 69
supported conversions, 70
implicit open
example, 9
implied unit numbers, 9
increasing formatted record size, 85
individual layer reference, 89
INQUIRE statement, 17
INQUIRE by file statement, 17
INQUIRE by unit statement, 17
internal file, 5
internal file identifier, 5
internal files
definition, 5
format, 6
standard Fortran, 5
introduction to FFIO
layered I/O, 55
layered I/O options, 59

L

layered I/O, 57
options, 59
overview, 55
specifying layers, 58
usage, 57
usage rules, 59
library buffer sizes, 85
library buffering, 53
library error messages
flexible file I/O error messages, 2
message system, 2
system error messages, 2
LISTIO_PRECISION, 15
local assign mode, 42
logical device
definition, 52

M

multitasking
standard Fortran I/O, 19
multithreading, 19

N

named pipes, 21
and binary data, 22
and EOF, 22
detecting EOF, 24
difference from normal files, 21
ISHELL call, 21
MAXPIPE parameter, 21
piped I/O example (EOF detection), 24
piped I/O example (no EOF detection), 22
receiving process
file structure, 22
restrictions, 21
sending process
file structure, 22
specifying file structure for binary data, 22
syntax, 21

with EOF detection
usage requirements, 24
namelist I/O, 15
null layer, 104

O

open processing, 31
and INQUIRE statement, 37
operations in FFIO, 114
optimization techniques, 79

P

performance enhancements, 64
permanent files
definition, 78
physical device I/O activities, 81
position property
definition, 8
Pthreads, 19

R

raw I/O, 53
read system call, 27
record blocking
removal, 64

S

sbin processing, 45
sequential access
external file properties, 6
setbuf function, 28
setvbuf function, 28
site layer, 108
standard error
unit number, 10
standard Fortran
EOF records, 8
standard input
unit number, 10
standard output
unit number, 10
stream
definition, 28
supported implicit data conversions, 70
syscall layer, 105
system cache, 53
definition, 52
system I/O, 27
asynchronous I/O, 27
synchronous I/O, 27
unbuffered I/O, 28
system layer, 106

T

temporary files
definition, 78
text file structure, 46
text files
and FFIO, 62
text layer, 106

U

u file processing, 46
unblocked data transfer
I/O layers, 64
unblocked file structure
and BACKSPACE statement, 44
and BUFFER IN/BUFFER OUT statements, 44
definition, 44
example, 38
specifications, 45

unblocked files
  and FFIO, 63
unbuffered I/O, 53
unformatted I/O, 16
UNIT intrinsic routine, 20
unit number
  standard error, 10
    access mode and form, 10
  standard input, 10
    access mode and form, 10
  standard output, 10
    access mode and form, 10
UNIX FFIO special files, 21
usage rules
  layered I/O options, 59
user layer, 108
user layer example, 116

V

valid unit numbers, 9
vms layer, 109

W

well-formed requests
  definition, 52
workstation and IEEE conversion, 74
write system call, 27