Application Programmer's I/O Guide

SG–2168 3.0.1

New Features

The following new features have been added for this revision of this manual:

- You can now specify numeric parameters for many FFIO layers with a keyword syntax. This new syntax is documented in Chapter 14.

- Information about support for I/O on IRIX systems has been added throughout this document.

- A new FFIO layer, the `global` layer, has been added and is documented in Chapter 14.

- Several new options were added to the `assign` command and are documented in Chapter 6.

- A new conversion routine, the `CRI2IBM` routine, has been added and is documented in Chapter 12.

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.

Contents

Preface

This publication describes Fortran input/output (I/O) techniques for use on Cray Research and on Silicon Graphics systems. It also contains information about advanced I/O topics such as asynchronous queued I/O and logical record I/O. Information about the interaction of the I/O library and the Cray Research Fortran 90 compiler, CF90, is also discussed. The information in this manual is pertinent for UNICOS systems, UNICOS/mk systems, and IRIX systems.

This document also serves as an I/O optimization guide for Fortran programmers on UNICOS systems. 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.

Information which is marked as available on IRIX systems is available with the MIPSpro 7 Fortran 90 compiler.

Related publications

The following Cray Research documents contain additional information that may be helpful:

- *Application Programmer's Library Reference Manual*, publication SR–2165

- *Optimizing Code on Cray PVP Systems*, publication SG–2192

- *Guide to Parallel Vector Applications*, publication SG–2182

- *UNICOS Performance Utilities Reference Manual*, publication SR–2040

- *UNICOS System Calls Reference Manual*, publication SR–2012

- *UNICOS System Libraries Reference Manual*, publication SR–2080

- *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*, publication SR–3907

- *CF90 Ready Reference*, publication SQ–3900

- *CF90 Commands and Directives Reference Manual*, publication SR–3901

- *Fortran Language Reference Manual, Volume 1*, publication SR–3902

- *Fortran Language Reference Manual, Volume 2*, publication SR–3903

- *Fortran Language Reference Manual, Volume 3*, publication SR–3905

Ordering Cray Research publications

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907, or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

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.

In addition to these formatting conventions, several naming conventions are used throughout the documentation. "Cray PVP systems" denotes all configurations of Cray parallel vector processing (PVP) systems which run the UNICOS operating system. "Cray MPP systems" denotes all configurations of the CRAY T3E series that run the UNICOS/mk operating system. "IRIX systems" denotes Silicon Graphics platforms that run the IRIX operating system.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `publications@cray.com`

- Contact your customer service representative and ask that an SPR or PV be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

Introduction [1]

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 for UNICOS and UNICOS/mk systems and for IRIX systems. This manual is for Fortran programmers who need general I/O information or who need information on how to optimize their I/O.

Some information in this manual addresses usage information for UNICOS and UNICOS/mk systems only. When this occurs, the information is flagged as applicable only to those systems.

This manual contains the following chapters:

- "Standard Fortran I/O," Chapter 2, page 5, discusses elements of the Fortran 90 standard that relate to I/O.

- "Fortran I/O Extensions," Chapter 3, page 21, discusses Cray Research extensions to the Fortran standard.

- "Tape and Named Pipe Support," Chapter 4, page 41, discusses tape handling and FIFO special files.

- "System and C I/O," Chapter 5, page 49, discusses system calls and Fortran callable entry points to C library routines.

- "The `assign` Environment," Chapter 6, page 55, 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.

- "File Structures," Chapter 7, page 73, discusses native file structures.

- "Buffering," Chapter 8, page 81, discusses file buffering as it applies to I/O.

- "Devices," Chapter 9, page 87, discusses types of storage devices.

- "Introduction to FFIO," Chapter 10, page 95, provides an overview of the Flexible File I/O system.

- "Using FFIO," Chapter 11, page 105, describes how to use FFIO with common file structures, and how to use FFIO to enhance program performance.

- "Foreign File Conversion," Chapter 12, page 125, discusses how to convert data from one file structure to another.

- "I/O Optimization," Chapter 13, page 155, discusses methods to speed up I/O processing.

- "FFIO Layer Reference," Chapter 14, page 183, provides details about individual FFIO layers.

- "Creating a `user` Layer," Chapter 15, page 231, provides an example of how to create an FFIO layer.

- "Older Data Conversion Routines," Appendix A, page 261, lists outdated data conversion routines.

1.1 The message system

The UNICOS operating system contains an error message system 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 on UNICOS and UNICOS/mk systems are within the range of 1000 to 2000. Library errors on IRIX systems 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`.

- On UNICOS systems, the tape system returns error numbers that are in the range of 90000 through 90500. The *UNICOS Tape Subsystem User's Guide*, publication SG–2051, lists tape system error messages.

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 (<e1> 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
```

Standard Fortran I/O [2]

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.

2.1 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 Section 2.2, 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.

2.1.1 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.

2.1.2 External files

In standard Fortran, one external unit may be connected to a file. Cray Research 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 on UNICOS and UNICOS/mk systems. On IRIX systems, the default direct access formatted file structure does not support this; the capability is available 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, page 55.

- **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/out or in/out list (*iolist*) item is processed. The use of a dollar sign ($) 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.

2.2 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).

- A Hollerith (integer) variable consisting of 1 to 8 left-justified, blank-filled or zero-filled ASCII characters. Each Hollerith unit identifier is associated with the file of the same name, which may exist. For example, unit `'red'L` is associated with the `red` file in the current working directory. The use of uppercase and lowercase characters is significant for file names. This extension is supported only on 64-bit systems.

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 workstation 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.

2.3 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.

On CRAY T3E systems with HPF_CRAFT, shared variables can be used in the I/O lists of formatted I/O, list-directed I/O, and unformatted I/O statements. Shared variables are not supported for the IOSTAT specifier, the unit specifier or any other control list specifier on I/O statements.

2.3.1 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.

2.3.1.1 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 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: 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 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 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 An). For Hollerith data in INTEGER variables, the optimal data edit descriptor is A8 (or R8).

2.3.1.2 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 90 `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.

YMP80   Causes list-directed and namelist output of real values to be of the format used in Cray Research's UNICOS 8.0 release and previous Cray Research library versions on UNICOS systems.

`LISTIO_OUTPUT_STYLE` provides a compatibility mode for the Cray Research CrayLibs 2.0 release and later versions. When set to `OLD`, this environment variable causes three effects:

- Repeated list-directed output values closely resemble those printed by the Cray Research CrayLibs 1.2 and prior releases. In these prior releases, the repeat counts never spanned vector array extents passed to the library from the compiler. In the current version of CrayLibs, the libraries coalesce repeat counts as much as possible to compress output and to ensure that compiler optimization does not affect the format of list-directed output.

- Value separators are not printed between adjacent nondelimited character values and noncharacter values printed by list-directed output in Fortran 90 files. In CrayLibs 2.0, the libraries produce one blank character as a value separator to comply with the ANSI Fortran 90 standard. No value separator is printed between adjacent nondelimited character values and noncharacter values in FORTRAN 77 files because the ANSI FORTRAN 77 standard requires that none be printed.

- A blank character will not be printed in column 1 when a list-directed statement with no I/O list items is executed. In the CrayLibs 2.0 release, the libraries started printing a blank character in column 1 to comply with the ANSI FORTRAN 77 and ANSI Fortran 90 standards.

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
```

2.3.1.2.1 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)
```

2.3.2 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
```

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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)
```

2.5 Private I/O on CRAY T3E systems

Private I/O consists of the READ, WRITE, OPEN, CLOSE, REWIND, ENDFILE, BACKSPACE, and INQUIRE statements. A private READ or WRITE statement is executed by the processing element (PE) that encounters it with no communication or coordination with other PEs.

At program start, unit numbers 0, 5, 6, and 100 through 102 are associated with stdin, stdout, and stderr. If stdin or stdout is not associated with a terminal, it is buffered. Results are unpredictable if more than one PE tries to read from units 5 or 100, or tries to write to units 6 or 101.

2.6 Multithreading and standard Fortran I/O

Multithreading is the concurrent use of multiple threads of control which operate within the same address space. On UNICOS systems, multithreading is available through macrotasking, Autotasking, and the Pthread interface. On UNICOS/mk systems, multithreading is available through the Pthreads interface. On IRIX systems, multithreading is available through DOACROSS compiler directives and through the Pthreads interface.

Standard Fortran I/O is thread-safe on UNICOS and IRIX systems. Standard Fortran I/O is not thread-safe on UNICOS/mk systems.

On UNICOS systems and IRIX systems, 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.

On UNICOS/mk systems (where Fortran I/O is not thread-safe), threaded programs must use locks or other synchronization around Fortran I/O statements to prevent concurrent execution of I/O statements on multiple threads. Failure to do so causes unpredictable results.

Fortran I/O Extensions   [3]

This chapter describes additional I/O routines and statements available with the CF90 compiler and the MIPSpro 7 Fortran 90 compiler. 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.

3.1 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. See Chapter 9, page 87, for more information about sector sizes.

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 COS blocked format is the default file structure on UNICOS and UNICOS/mk systems for files (not tapes) that are opened explicitly as unformatted sequential or implicitly by a BUFFER IN or BUFFER OUT statement. The F77 format is the default file structure on IRIX systems.

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.

3.1.1 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)
```

3.1.2 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)
```

3.1.3 Positioning (deferred implementation on IRIX systems)

The GETPOS(3F) and SETPOS(3F) file positioning routines change or indicate the position of the current file. The GETPOS routine returns the current position of a file being read. The SETPOS routine positions a tape or mass storage file to a previous position obtained through a call to GETPOS.

You can use the GETPOS and SETPOS positioning statements on buffer I/O files. These routines can be called for random positioning for BUFFER IN and BUFFER OUT processing. These routines can be used with COS blocked files on disk, but not with COS blocked files on tape.

You can also use these routines with the standard Fortran READ and WRITE statements. The direct-access mode of standard Fortran is an alternative to the GETPOS and SETPOS functionality.

3.2 Random access I/O routines (not available on IRIX systems)

The record-addressable random-access file I/O routines let you generate variable length, individually addressable records. The I/O library updates indexes and pointers.

Each record in a random-access file has a 1-word (64-bit) key or number indicating its position in an index table of records for the file. This index table contains a pointer to the location of the record on the device and can also contain a name of each record within the file.

Alphanumeric record keys increase CPU time compared to sequential integer record keys because the I/O routines must perform a sequential lookup in the index array for each alphanumeric key. Each record should be named a numeric value *n*; *n* is the integer that corresponds to the *n* th record created on the file.

The following two sets of record-addressable random-access file I/O routines are available:

- The Mass Storage (MS) package provides routines that perform buffered, record-addressable file I/O with variable-length records. It contains the OPENMS, READMS, WRITMS, CLOSMS, WAITMS, FINDMS, SYNCMS, ASYNCMS, CHECKMS, and STINDX routines.

- The Direct Random (DR) package provides routines that perform unbuffered, record-addressable file I/O. It contains the OPENDR, READDR, WRITDR, CLOSDR, WAITDR, SYNCDR, ASYNCDR, CHECKDR, and STINDR routines. The amount of data transferred for a record is rounded up to a multiple of 512 words, because I/O performance is improved for many disk devices.

Both synchronous and asynchronous MS and DR I/O can be performed on a random-access file. You can use these routines in the same program, but they must not be used on the same file simultaneously. The MS and DR packages cannot be used for tape files.

If a program uses asynchronous I/O, it must ensure that the data movement is completed before trying to access the data. Because asynchronous I/O has a larger overhead in CPU time than synchronous I/O, only very large data transfers should be done with asynchronous I/O. To increase program speed, the program must be able to do a significant amount of CPU-intensive work or other I/O while the asynchronous I/O is executing.

The MS library routines are used to perform buffered record-addressable random-access I/O. The DR library routines are used to perform unbuffered record-addressable random-access I/O.

These library routines are not internally locked to ensure single-threading; a program must lock each call to the routine if the routine is called from more than one task.

The following list describes these two packages in more detail. For details about the routines discussed in this section, see the individual man pages for each routine.

- OPENMS(3F) and OPENDR(3F) open a file and specify the file as a random-access file that can be accessed by record-addressable random-access I/O routines.

  These routines must be used to open a file before the file can be accessed by other MS or DR package routines. OPENMS sets up an I/O buffer for the random-access file. These routines read the index array for the file into the array provided as an argument to the routine. CLOSMS or CLOSDR must close any files opened by the OPENMS or OPENDR routine. The following are examples of these two routines:

  ```
  CALL OPENMS(80,intarr,len,it,ierr)
  CALL OPENDR(20,inderr,len,itflg,ierr)
  ```

- READMS(3F) performs a read of a record into memory from a random-access file. READDR reads a record from a random-access file into memory.

  If READDR is used in asynchronous mode and the record size is not a multiple of 512 words, user data can be overwritten and not restored. You can use SYNCDR to switch to a synchronous read; the data is copied and restored after the read has completed. The following are examples of these routines:

```
CALL READMS(80,ibuf,nwrd,irec,ierr)
CALL READDR(20,iloc,nwrd,irec,ierr)
```

- WRITMS(3F) writes to a random-access file on disk from memory. WRITDR writes data from user memory to a record in a random-access file on disk. Both routines update the current index. The following are examples of these routines:

```
CALL WRITMS(20,ibuf,nwrd,irec,irflg,isflag,ierr)
CALL WRITDR(20,ibuf,nwrd,irec,irflag,isflg,ierr)
```

- The CLOSMS(3F) and CLOSDR routines write the master index specified in the call to OPENMS or OPENDR from the array provided in the user program to the random-access file and then close the file. These routines also write statistics about the file to the stderr file. The following are examples of these routines:

```
CALL CLOSMS(20,ierr)
CALL CLOSDR(20,ierr)
```

- ASYNCMS(3F) and ASYNCDR set the I/O mode for the random-access routines to asynchronous. I/O operations can be initiated and subsequently proceed simultaneously with the actual data transfer. If the program uses READMS, precede asynchronous reads with calls to FINDMS. The following are examples of these routines:

```
CALL ASYNCMS(20,ierr)
CALL ASYNCDR(20,ierr)
```

- CHECKMS(3F) and CHECKDR check the status of the asynchronous random-access I/O operation. The following are examples of these routines:

```
CALL CHECKMS(20,istat,ierr)
CALL CHECKDR(20,istat,ierr)
```

- WAITMS(3F) and WAITDR wait for the completion of an asynchronous I/O operation. They return a status flag indicating if the I/O on the specified file completed without error. The following are examples of these routines:

```
CALL WAITMS(20,istat,ierr)
CALL WAITDR(20,istat,ierr)
```

- SYNCMS(3F) and SYNCDR set the I/O mode for the random-access routines to synchronous. All future I/O operations wait for completion. The following are examples of these routines:

```
CALL SYNCMS(20,ierr)
CALL SYNCDR(20,ierr)
```

- STINDX(3F) and STINDR allow an index to be used as the current index by creating a subindex. These routines reduce the amount of memory needed by a file that contains a large number of records. They also maintain a file containing records logically related to each other. Records in the file, rather than records in the master index area, hold secondary pointers to records in the file.

  These routines allow more than one index to manipulate the file. Generally, STINDX or STINDR toggle the index between the master index maintained by OPENMS/OPENDR and CLOSMS/CLOSDR and the subindex supplied by the Fortran program. The following are examples of these routines:

  ```
  CALL STINDX(20,inderr,len,itflg,ierr)
  CALL STINDR(20,inderr,len,itflg,ierr)
  ```

- FINDMS(3F) asynchronously reads the desired record into the data buffers for the specified file. The next READMS or WRITMS call waits for the read to complete and transfers data appropriately. An example of a call to FINDMS follows:

  ```
  CALL FINDMS(20,inwrd,irec,ierr)
  ```

The following program example uses the MS package:

Example 1: MS package use

```
program msio
dimension r(512)
dimension idx(512)
data r/512*2.0/
irflag=0

call openms(1,idx,100,0,ier)

do 100 i=1,100
  call writms(1,r,512,i,irflag,0,ier)
  if(ier.ne.0)then
    print *,"error on writms=",ier
    goto 300
  end if
100    continue
```

```
          do 200 i=1,100
            call readms(1,r,512,i,irflag,0,ier)
            if(ier.ne.0)then
              print *,"error on readms=",ier
              goto 300
            end if

200     continue
300     continue
        call closms(1,ier)
        end
```

The following program uses the DR package:

Example 2: DR package use

```
        program daio
        dimension r(512)
        dimension idx(512)
        data r/512*2.0/
        irflag=0
        ierrs=0

        call assign('assign -R',ier1)
        call asnunit(1,'-F mr.save.ovf1:10:200:20',ier2)
        if(ier1.ne.0.or.ier2.ne.0)then
            print *,"assign error=",ier1,ier2
            ierrs=ierrs+1
        end if

        call opendr(1,idx,100,0,ier)
        if(ier.ne.0)then
          print *,"error on opendr=",ier
          ierrs=ierrs+1
        end if

        do 100 i=1,100
          call writdr(1,r,512,i,irflag,0,ier)
          if(ier.ne.0)then
            print *,"error on writdr=",ier
            ierrs=ierrs+1
          end if
100     continue
```

```
                     do 200 i=1,100
                       call readdr(1,r,512,i,irflag,0,ier)
                       if(ier.ne.0)then
                          print *,"error on readdr=",ier
                          ierrs=ierrs+1
                       end if
200     continue
300     call closdr(1,ier)
        if(ier.ne.0)then
          print *,"error on readdr=",ier
          ierrs=ierrs+1
        end if
400     continue
        if(ierrs.eq.0)then
          print *,"daio passed"
        else
          print *,"daio failed"
        end if
        end
```

3.3 Word-addressable I/O routines (not available on IRIX systems)

A word-addressable (WA) random-access file consists of an adjustable number of contiguous words. The WA package performs unformatted buffered I/O; the WA routines perform efficiently when the I/O buffers are set to a size large enough to hold several records that are frequently read and/or written. When a WA read operation is executed, the I/O buffers are searched to see if the data that will be read is already in the buffers. If the data is found in the I/O buffers, I/O speedup is achieved because a system call is not needed to retrieve the data.

A program using the package may access a word or a contiguous sequence of words from a WA random-access file. The WA package cannot be used for tape files.

Although the WA I/O routines provide greater control over I/O operations than the record-addressable routines, they require that the user track information that the system would usually maintain when other forms of I/O are used. The program must keep track of the word position of each record in a file that it will read or write with WA I/O. This is easiest to do with fixed-length records; with variable-length records, the program must store record lengths for the file where they can be retrieved when the file is accessed. When variable-length records are used, the program should use record–addressable I/O.

The WA package allows both synchronous and asynchronous I/O. To speed up the program, the program must be able to do a significant amount of CPU-intensive work or other I/O while the asynchronous I/O is executing.

These library routines are not internally locked to ensure single-threading; a program must lock each call to the routine if the routine is called from more than one task.

The following list briefly describes the routines in this package; for details about the routines discussed in this section, see the individual man pages for each routine.

- WOPEN(3F) opens a file and specifies it as a word-addressable random-access file. WOPEN must be called before any other WA routines are called because it creates the I/O buffer for the specified file by using blocks. By using WOPEN, you can combine synchronous and asynchronous I/O to a given file while the file is opened. The following is an example of a call to WOPEN:

  ```
  CALL WOPEN(30,iblks,istat,err)
  ```

- GETWA(3F) synchronously reads data from a buffered word-addressable random-access file. SEEK(3F) is used with GETWA to provide more efficient I/O; the SEEK routine performs an asynchronous pre-fetch of data into a buffer. The following is an example of a call to GETWA:

  ```
  CALL GETWA(30,iloc,iadr,icnt,ierr)
  ```

- SEEK(3F) asynchronously reads data from the word-addressable file into a buffer. A subsequent GETWA call will deliver the data from the buffer to the user data area. This provides a way for the user to do asynchronous read-ahead. The following is an example of a call to SEEK:

  ```
  CALL SEEK(30,iloc,iadr,icnt,ierr)
  ```

- PUTWA(3F) synchronously writes from memory to a word-addressable random-access file. The following is an example of a call to PUTWA:

  ```
  CALL PUTWA(30,iloc,iadr,icnt,ierr)
  ```

  APUTWA(3F) asynchronously writes from memory to a word-addressable random-access file. The following is an example of a call to APUTWA:

  ```
  CALL APUTWA(30,iloc,iadr,icnt,ierr)
  ```

- WCLOSE(3F) finalizes changes and additions to a WA file and closes it. The following is an example of a call to WCLOSE:

```
          CALL WCLOSE(30,ierr)
```

The following is an example of a program which uses the WA I/O routines:

Example 3: WA package use

```
          program waio
          dimension r(512), r1(512)
          iblks=10               !use a 10 block buffer
          istats=1               !print out I/O Stats

          call wopen(1,iblks,0,ier)
          if(ier.ne.0)then
            print *,"error on wopen=",ier
            goto 300
          end if

          iaddr=1
          do 100 k=1,100

            do 10 j=1,512
10          r(j)=j+k
            call putwa(1,r,iaddr,512,ier)
            if(ier.ne.0)then
               print *,"error on putwa=",ier," rec=",k
               goto 300
            end if
            iaddr=iaddr+512
100       continue

          iaddr=1
          do 200 k=1,100
            call getwa(1,r1,iaddr,512,ier)
            if(ier.ne.0)then
               print *, "error on getwa=",ier," rec=",k
               goto 300
            end if
            iaddr=iaddr+512
200       continue
300       continue
          call wclose(1)
          end
```

3.4 Asynchronous queued I/O (AQIO) routines (not available on IRIX systems)

The asynchronous queued I/O (AQIO) routines perform asynchronous, queued I/O operations. Asynchronous I/O allows your program to continue executing while an I/O operation is in progress, and it allows several I/O requests to be active concurrently. AQIO further refines asynchronous I/O by allowing a program to queue several I/O requests and to issue one request to the operating system to perform all I/O operations. When queuing I/O requests, the overhead that is associated with calling the operating system is incurred only once per group of I/O requests rather than once per request as with other forms of I/O.

AQIO also offers options for streamlining I/O operations that involve fixed-length records with a fixed-skip increment through the user file and a fixed-skip increment through program memory. A form of this is a read or write that involves contiguous fixed-length records. Such an operation is called a *compound AQIO request* or a *compound AQIO operation.* AQIO provides separate calls for compound operations so that a program can specify multiple I/O operations in one call, thus saving I/O time.

Asynchronous I/O has a larger overhead in system CPU time than synchronous I/O; therefore, only large data transfers should be done using asynchronous I/O. To speed up the program, the program must be able to do a significant amount of CPU-intensive work or other I/O while the asynchronous I/O is executing.

The value of the *queue* argument on the AQWRITE/AQWRITEC(3F) or AQREAD/AQREADC(3F) call controls when the operating system is called to process the request. If *queue* is nonzero, packets are queued in the AQIO buffer and the operating system is not called to start packet processing until the buffer is full (for example, to queue 20 packets, the program would issue 19 AQWRITE calls with *queue* set to a nonzero value and then set it to 0 on the twentieth call).

On CRAY T3E systems, when a program opens a file with AQOPEN, a file handle is returned. The library associates this handle with information in the processing element's (PE) local memory; therefore, the file handle should not be used by other PEs. More than one PE can open a file with AQOPEN; if coordination between the different PEs is required, the user must do the coordination using synchronization routines.

The following list briefly describes the AQIO routines; for details about the routines discussed in this section, see the individual man pages for each routine.

- AQOPEN(3F) opens a file for AQIO. The AQOPEN call must precede all other AQIO requests in a Fortran program.

- `AQCLOSE`(3F) closes an AQIO file.

- The `AQREAD` function queues a simple asynchronous I/O `read` request.

- `AQREADC`(3F) lets you use a compound AQIO request call to transfer fixed-length records repeatedly. You must provide the values for a repeat count, memory skip increment, and disk increment arguments. `AQREADC` transfers the first record from disk and increments the starting disk block and the starting user memory by the amounts you specify.

  To transfer data to a continuous array in memory, set the memory skip increment value to the record length in words. To transfer data sequentially from disk, set the disk increment value to the record length in blocks. See Example 4, page 33, for an example of a program using AQIO read routines.

- `AQWRITE` queues a simple asynchronous `write` request.

- `AQWRITEC` provides a compound AQIO request call when repeatedly transferring fixed-length records. The program supplies the repetition count, the disk skip increment, and the memory skip increment on these compound AQIO calls.

  AQIO then transfers the first record to or from disk and increments the starting disk block and the starting user memory address. To transfer data from a contiguous array in memory, set the memory skip increment value to the record length in words. To transfer data sequentially to disk, set the disk increment value to the record length in blocks.

- `AQSTAT` checks the status of AQIO requests. `AQWAIT` forces the program to wait until all queued entries are completed.

  After queuing a `AQWRITE` or `AQREAD` request and calling the operating system, you may need to monitor their completion status to know when it is safe to use the data or to reuse the buffer area. `AQSTAT` returns information about an individual AQIO request.

  The *reqid* argument of `AQREAD`/`AQREADC` and `AQWRITE`/`AQWRITEC` is stored in the packet buffer and can be used in an `AQSTAT` call to monitor the completion status of a particular transfer. The *aqpsize* argument to `AQOPEN` is important because of the ability to monitor the status.

  A requested ID can be deleted after the request completes but before its status is checked because each request buffer is reused. This can happen, for example, if you set the *aqpsize* argument in `AQOPEN` to be 20, and issued 30 requests. If you then request the status of the first request, `AQSTAT` returns 0, indicating that the requested ID was not found.

3.4.1 Error detection by using AQIO

Because of the asynchronous nature of AQIO, error detection and reporting with AQIO may not occur immediately on return from a call to an asynchronous queued I/O subroutine. If one of the queued I/O requests causes an error when the operating system tries to do the I/O, the error is returned in a subsequent AQIO request.

For example, if a program issues an AQWRITE with *queue* set to 0, I/O is initiated. If no previous errors occurred, a 0 status is returned from this statement even though this request may ultimately fail. If the request fails, for example, because it tried to exceed the maximum allowed file size, the error is returned to the user in the subsequent AQIO statement that coincides with its detection. If the next AQIO statement is AQWAIT, the error is detected and returned to the user. If the next AQIO statement is AQSTAT, the error is detected and reported only if the requested ID failed. When an error is reported to the user, it is not reported again. Checking the status after each AQIO statement ensures that the user program detects all errors.

Example 4: AQIO routines: compound read operations

```
      PROGRAM AQIO1
      IMPLICIT INTEGER(A-Z)
      PARAMETER (TOTREQ=20)
      PARAMETER (AQPSIZE=20)
      INTEGER AQP
      INTEGER BUFFER (TOTREQ*512)
      INTEGER EVNBUF (TOTREQ/2*512)
      INTEGER ODDBUF (TOTREQ/2*512)

      CALL AQOPEN (AQP,AQPSIZE,'FILE4'H,STAT)
      IF (STAT.NE.0) THEN
         PRINT *,'AQOPEN FAILED, STATUS= ',STAT
         CALL ABORT()
      ENDIF

C     INITIALIZE DATA
      DO 10 I=1,TOTREQ*512
         BUFFER(i) = I
10    CONTINUE

      DO 50 RNUM=1,TOTREQ
C       QUEUE THE REQUESTS
C         INITIATE I/O ON THE LAST REQUEST
```

```
C        THE DATA FROM BUFFER IS WRITTEN IN A SEQUENTIAL
C        FASHION TO DISK
         QUEUE=1
         IF (RNUM.EQ.TOTREQ) QUEUE=0
            OFFSET= (RNUM-1)*512+1
          CALL AQWRITE(
         '    AQP,
         '    BUFFER(OFFSET),   !start address
         '    RNUM-1,           !block address
         '    1,                !number of blocks
         '    RNUM,             !request id
         '    QUEUE,            !queue request or start I/O
         '    STAT)             !return status
         IF (STAT.NE.0)THEN
            PRINT*,'AQWRITE FAILED, STATUS= ',STAT
            CALL ABORT()
         ENDIF
50     CONTINUE

C      WAIT FOR I/O TO COMPLETE
       CALL AQWAIT (AQP,STAT)
       IF (STAT.LT.0) THEN
          PRINT*,'AQWAIT AFTER AQWRITE FAILED, STATUS=',STAT
          CALL ABORT()
       ENDIF

C      NOW ISSUE TWO COMPOUND READS.  THE FIRST READ
C      GETS THE ODD SECTORS AND THE SECOND GETS THE
C      EVEN SECTORS.
C
       INCS=TOTREQ/2-1
       CALL AQREADC(
       '    AQP,
       '    ODDBUF(1),         ! start address
       '    512,               ! mem stride
       '    1,                 ! block number
       '    1,                 ! number of blocks
       '    2,                 ! disk stride
       '    INCS,              ! incs
       '    1,                 ! request id
       '    1,                 ! queue request
       '    STAT1)             ! return status
```

```
            CALL AQREADC(
      '     AQP,
      '     EVNBUF(1),         ! start address
      '     512,               ! mem stride
      '     0,                 ! block number
      '     1,                 ! number of blocks
      '     2,                 ! disk stride
      '     INCS,              ! incs
      '     2,                 ! request id
      '     0,                 ! start request
      '     STAT2)             ! return status
            IF ((STAT1.NE.0). OR. (STAT2.NE.0)) THEN
               PRINT *,'AQREADC FAILED, STATUS= ',STAT1,STAT2
               CALL ABORT()
            ENDIF

            CALL AQWAIT (AQP,STAT)
            IF (STAT.LT.0) THEN
               PRINT *,'AQWAIT FAILED, STATUS= ',STAT
               CALL ABORT()
            ENDIF

      C     VERIFY THAT THE DATA READ WAS CORRECT
            K = 1
            DO 90 I = 1,TOTREQ,2
               DO 80 J = 1,512
                 IF (EVNBUF (J+(K-1)*512).NE.J+(I-1)*512)THEN
                     PRINT *,'BAD DATA EVN',EVNBUF(J+(K-1)*512),J,I,K
                     CALL ABORT()
                 ENDIF
      80      CONTINUE
               K=K+1
      90    CONTINUE
            K = 1
            DO 99 I = 2,TOTREQ,2
               DO 95 J = 1,512
                 IF (ODDBUF(J+(K-1)*512).NE.J+(I-1)*512)
                     PRINT *,'BAD DATA ODD',ODDBUF(J+(K-1)*512),J,I,K
                     CALL ABORT()
                 ENDIF
      95      CONTINUE
               K=K+1
      99    CONTINUE
```

```
              CALL AQCLOSE(AQP,STAT)
              IF(STAT.NE.0) THEN
                      PRINT *,'AQCLOSE FAILED, STATUS= ',STAT
                      CALL ABORT()
                   ENDIF
             END
```

Example 5: AQIO routines: error detection

```
    PROGRAM AQIO2
    IMPLICIT INTEGER(A-Z)
    PARAMETER (TOTREQ=20)
    PARAMETER (AQPSIZE=20)
    INTEGER AQP
    INTEGER BUFFER (TOTREQ*512)
    INTEGER INBUF (512)

    CALL AQOPEN (AQP,AQPSIZE,'FILE4'H,STAT)
    IF (STAT.NE.0) THEN
       PRINT *,'AQOPEN FAILED, STATUS=',STAT
       CALL ABORT()
    ENDIF

    DO 50 RNUM=1,TOTREQ

C    QUEUE THE REQUESTS
C    INITIATE I/O ON THE LAST REQUEST
C    THE DATA FROM BUFFER WILL BE WRITTEN IN A
C    SEQUENTIAL FASHION TO DISK
    QUEUE=1
    IF (RNUM.EQ.TOTREQ) QUEUE=0
        OFFSET= (RNUM-1)*512+1
        CALL AQWRITE (
        '    AQP,
        '    BUFFER (OFFSET),   ! start address
        '    RNUM-1,            ! block number
        '    1,                 ! number of blocks
        '    RNUM,              ! request id
        '    QUEUE,             ! queue request or start I/O
        '    STAT)              ! return status
    IF (STAT.NE.0) THEN
        PRINT *,'AQWRITE FAILED, STATUS=',STAT
        CALL ABORT ()
```

```
            ENDIF
     50 CONTINUE

        CALL AQWAIT (AQP,STAT)
        IF (STAT.LT.0) THEN
          PRINT *,'AQWAIT AFTER AQWRITE FAILED, STATUS= ',STAT
          CALL ABORT ()
        ENDIF

C  NOW WE ISSUE A READ. TO ILLUSTRATE ERROR
C  DETECTION, WE ATTEMPT TO READ BEYOND THE END
C  OF THE FILE
        CALL AQREAD (
       '      AQP,
       '      INBUF(1),     ! start address
       '      TOTREQ+1,     ! block number
       '      1,            ! number of blocks
       '      TOTREQ+1,     ! request id
       '      0,            ! start I/O
       '      STAT)         ! return status

        IF (STAT.NE.0)THEN
          PRINT *,'AQREAD FAILED, STATUS=',STAT
          CALL ABORT()
        ENDIF

        CALL AQWAIT (AQP,STAT)
C  BECAUSE WE ATTEMPTED TO READ BEYOND THE END
C  OF THE FILE, AQWAIT WILL RETURN A NEGATIVE
C  VALUE IN "STAT", AND THE PROGRAM WILL ABORT IN
C  THE FOLLOWING STATEMENT

        IF (STAT.LT.0) THEN
          PRINT *,'AQWAIT AFTER AQREAD FAILED, STATUS= ',STAT
          CALL ABORT()
        ENDIF

        CALL AQCLOSE (AQP,STAT)
        IF (STAT.NE.0) THEN
          PRINT *,'AQCLOSE, STATUS= ',STAT
          CALL ABORT()
        ENDIF
        END
```

The following is the output from running this program:

```
AQWAIT AFTER AQREAD FAILED, STATUS= -1202
```

3.5 Logical record I/O routines (Not available on IRIX systems)

The logical record I/O routines provide word or character granularity during read and write operations on full or partial records. The read routines move data from an external device to a user buffer. The write routines move data from a user buffer to an external device.

The following list briefly describes these routines; for details about the routines discussed in this section, see the individual man pages for each routine.

- READ and READP move words of data from disk or tape to a user data area. READ(3F) reads words in full-record mode. READP reads words in partial-record mode.

  READ positions the file at the beginning of the next record after a READ. READP positions the file at the beginning of the next word in the current record after a READP. If foreign record translation is enabled for the specified unit, the bits from the foreign logical records are moved without conversion. Therefore, if the file contained IBM data, that data is not converted before it is stored. The following are examples of calls to READ and READP:

  ```
  CALL READ (7,ibuf,icnt,istat,iubc)
  CALL READP(8,ibuf,icnt,istat,iubc)
  ```

- READC(3F) reads characters in full-record mode. READCP reads characters in partial-record mode. Characters are moved to the user area with only one character per word and are right-justified in the word. The bits from foreign logical records are moved after conversion when foreign record translation is enabled for the specified unit. The following are examples of calls to READC and READCP:

  ```
  CALL READC (9,ibuf,icnt,istat)
  CALL READCP (10,ibuf,icnt,istat)
  ```

- READIBM(3F) reads IBM 32-bit floating-point words that are converted to Cray 64-bit words. The IBM 32-bit format is converted to the equivalent Cray 64-bit value and the result is stored. A conversion routine, IBM2CRAY(3F), converts IBM data to Cray format. A preferred method to obtain the same result is to read the file with an unformatted READ

statement and then convert the data through a call to IBM2CRAY. The
following is an example of a call to READIBM:

```
CALL READIBM (7,ibuf,ileng,incr)
```

- WRITE(3F) writes words in full-record mode. WRITEP writes words in
  partial-record mode. WRITE and WRITEP move words of data from the user
  data area to an I/O buffer area. If foreign record translation is enabled, no
  data conversion occurs before the words are stored in the I/O buffer area.
  The following are examples of calls to WRITE and WRITEP:

```
CALL WRITE (8,ibuf,icnt,iubc,istat)
CALL WRITEP (9,ibuf,icnt,iubc,istat)
```

- WRITEC(3F) writes characters in full-record mode. WRITECP writes
  characters in partial-record mode. Characters are packed into the buffer for
  the file. If foreign record translation is enabled, the characters are converted
  and then packed into the buffer. The following are examples of calls to
  WRITEC and WRITECP:

```
CALL WRITEC (10,icbuf,iclen,istat)
CALL WRITECP (11,icbuf,iclen,istat)
```

- WRITIBM(3F) writes Cray 64-bit values as IBM 32-bit floating-point words.
  The Cray 64-bit values are converted to IBM 32-bit format, using a
  conversion routine, CRAY2IBM(3F). After this conversion, you can use an
  unformatted WRITE statement to write the file. The following is an example
  of the call to WRITIBM:

```
CALL WRITIBM (12,ibuf,ilen,incr)
```

Tape and Named Pipe Support   [4]

Tape handling is usually provided through the tape subsystem with a minimum of user intervention. However, user end-of-volume (EOV) processing, bad data handling, and some tape positioning actions require additional support routines.

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.

The er90 and tape layers are not available on IRIX systems. The er90 layer is not available on CRAY T3E systems.

4.1 Tape support (not available on IRIX systems)

You can write and read from a tape using formatted or unformatted I/O statements. You can also use BUFFER IN and BUFFER OUT statements and the logical record routines (READC, READP, WRITEC, and WRITEP) to access the tape file from a Fortran program. For complete details about using tape files in Fortran programs on UNICOS and UNICOS/mk platforms, see the Cray document, *UNICOS Tape Subsystem User's Guide*, publication SG–2051.

4.1.1 User EOV processing

Several library routines assist users with EOV processing from a Fortran program. Tape-volume switching is usually handled by the tape subsystem and is transparent to the user. However, when a user requests EOV processing, the program gains control at the end of tape, and the program may perform special processing. The following library routines can be used with tape processing:

- CHECKTP(3F) checks the tape position.

- CLOSEV(3F) closes the volume and mounts the next volume in a volume identifier list.

- ENDSP(3F) disables special tape processing.

- SETSP(3F) enables and disables EOV processing.

- STARTSP(3F) enables special tape processing.

4.1.2 Handling bad data on tapes

The SKIPBAD(3F) and ACPTBAD(3F) routines can be called from a Fortran program to handle bad data on tape files.

- SKIPBAD skips bad data; it does not write it to the buffer.

- ACPTBAD makes bad data available by transferring it to the user-specified buffer. It allows a program to read beyond bad data within a file by moving it into the buffer and positioning past the bad data.

4.1.3 Positioning

The GETTP(3F) and SETTP(3F) file positioning routines change or indicate the position of the current file.

- GETTP gets information about an opened tape file.

- SETTP positions a tape file at a tape block and/or a tape volume.

4.2 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. Users with the MIPSpro 7 Fortran 90 compiler can use the `ACTION=READ` specifier on the `OPEN` statement to open a file as read-only.

4.2.1 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 6: 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 7: 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)+1
10    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
```

4.2.2 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. Users with the MIPSpro 7 Fortran 90 compiler can use the ACTION=READ specifier on the OPEN statement to open a file as read-only.

4.2.3 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 8: 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 9: 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
```

System and C I/O [5]

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 and describes C I/O on UNICOS/mk systems.

5.1 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. For more information about the system calls described in this chapter, see the Cray document, *UNICOS System Calls Reference Manual*, publication SR–2012 or the individual man pages.

5.1.1 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.

5.1.2 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.

On UNICOS and UNICOS/mk systems, the READA(3F) and WRITEA(3F) functions provide a Fortran interface to the reada(2) and writea(2) system calls. The reada system call reads a specified number of bytes from a file into a specified buffer. The system call returns immediately, even if the data cannot be delivered until later. The writea system call writes from a buffer to a file as specified.

5.1.3 `listio` I/O (not available on IRIX systems)

Use the listio(2) system call to initiate a list of distinct I/O requests and, optionally, wait for all of them to complete. No subroutine or function interface to listio exists in Fortran. The AQIO package provides an indirect Fortran interface to listio.

5.1.4 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. On UNICOS and UNICOS/mk systems, doing asynchronous system buffered I/O (for example, not using O_RAW) can cause performance problems because system caching can cause performance problems.

5.2 C I/O

This section describes C library I/O from Fortran, and describes C library I/O on CRAY T3E systems.

5.2.1 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. UNICOS and UNICOS/mk systems also provide Fortran callable entry points to many of these routines. For more information about the C library functions, see the Cray document, *UNICOS System Libraries Reference Manual*, publication SR–2080.

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 *stream* 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.

On UNICOS and UNICOS/mk systems, Fortran interfaces exist for the following C routines that use the `FILE` type:

FCLOSE	FPUTS
FDOPEN	FREAD
FGETS	FREOPEN
FILENO	FSEEK
FOPEN	

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 10: 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
```

5.2.1.1 C I/O on CRAY T3E systems

When using system calls on CRAY T3E systems, if more than one processing element (PE) opens the same file with an open(2) system call, distinct file descriptors are returned. If each PE uses its file descriptor to perform a read operation on the file, each PE reads the entire file.

If each PE uses its file descriptor to perform a write operation to the file, the results are unpredictable.

When a program opens a stream with `fopen()`, a pointer to the `stdio.h` file structure associated with the stream is returned. This stream pointer points to a structure contained in local memory on a PE; therefore, the stream pointer may not be used from another PE. If a stream is buffered, its buffer is contained in local memory to the PE that opened it, and it is unknown to other PEs.

At program startup, each PE has an open `stdio` stream pointer for `stdin`, `stdout`, and `stderr`; `stderr` is usually not fully buffered and `stdin` and `stdout` are fully buffered only if they do not refer to an interactive device. Buffers associated with `stdin`, `stdout`, and `stderr` are local to a PE.

Results are unpredictable if `stdin` is buffered and more than one PE attempts to read from it and if `stdout` is buffered and more than one PE attempts to write to it. The file descriptor for any of these streams is shared across all PEs; therefore, applying an `fclose ()` operation to `stdin`, `stdout`, or `stderr` on any PE closes that stream on all PEs.

When a program opens a file for flexible file input/output (FFIO) with `ffopen`(3C) or `ffopens`(3C), the library associates a structure local to the PE that contains descriptive data with the value returned to the user. Therefore, the value returned by `ffopen` may not be used from another PE. The FFIO processing layers may also contain buffering that is local to the PE. Attempting to perform an `ffopen` operation and do I/O to the same file from more than one PE may produce unpredictable results.

The `assign` Environment [6]

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.

On IRIX systems, the `assign` command affects Fortran programs compiled with the MIPSpro 7 Fortran 90 or programs compiled with the MIPSpro 7.2 F77 compiler and the `-craylibs` compiler option. It also affects programs that use `ffopen`(3C).

This chapter describes the `assign`(1) command and the `ASSIGN`(3F) library routine, which are used for these purposes.

6.1 `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.

6.1.1 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`, `g:aq`, 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.

6.1.2 The `assign` command

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

UNICOS and UNICOS/mk systems:

```
assign [-I] [-O] [-a actualfile] [-b bs] [-c] [-d bdr] [-f fortstd]
    [-l buflev] [ -m setting] [-n sz[:st]] [-p partlist] [-q ocblks] [-r setting]
    [-s ft] [-t] [-u bufcnt] [-w setting] [-x setting] [-y setting] [-C charcon]
    [-D fildes] [-F spec[,specs]] [-L setting] [-N numcon] [-P scope] [-R]
    [-S setting] [-T setting] [-U setting] [-V] [-W setting] [-Y setting]
    assign_object
```

IRIX systems:

```
assign [-a actualfile] [-b bs] [-f fortstd] [-s ft] [-t]  [-y setting]
    [-B setting] [-D fildes] [-F spec[,specs]] [-I] [-O] [-R] [-S setting]
    [-T setting] [-U setting] [-V] [-W setting] [-Y 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. The `assign` command is
implemented through the `ASSIGN`(3F), `ASNFILE`(3F), and `ASNUNIT`(3F) routines
for Cray Research Programming Environment releases prior to 1.2.

The following are the `assign` command control options:

-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.

-c   Contiguous storage. Must be used with the `-n`
                         option. Deferred implementation on IRIX
                         systems.

-d *bdr*	Online tape bad data recovery. Specify either `skipbad` or `acptbad` for *bdr*. Deferred implementation on IRIX systems.
-f *fortstd*	Fortran standard.
	Specify `77` to be compatible with the FORTRAN 77 standard and Cray Research's CF77 compiling system.
	Specify `90` to be compatible with the Fortran 90 standard and Cray Research's CF90 compiling system.
	Specify `irixf77` to be compatible with Silicon Graphic's FORTRAN 77 compiling system which runs on IRIX systems.
	Specify `irixf90` to be compatible with the MIPSpro 7 Fortran 90 compiler.
-l *buflev*	Kernel buffering. Specify `none`, `ldcache`, or `full` for *buflev*. If this is not set, the level of buffering is dependent on the type of open operation being performed. Deferred implementation on IRIX systems.
-m *setting*	Special handling of a direct access file that will be accessed concurrently by several processes or tasks. Special handling includes skipping the check that only one Fortran unit be connected to a unit, suppressing file truncation to true size by the I/O buffering routines, and ensuring that the file is not truncated by the I/O buffering routines. Enter either `on` or `off` for *setting*. Not available on IRIX systems.
-n *sz* [:*st*]	Amount of system file space to reserve for a file. This is a number of 4096–byte blocks. Used by Fortran I/O and auxiliary I/O (AQIO, WAIO, DRIO, and MSIO). The optional *st* value is an obsolete way to specify the -q assign attribute. Use of -q is preferable to using the *st* value on -n. Deferred implementation on IRIX systems.

-p *partlist*	File system partition list. Used by Fortran I/O and auxiliary I/O. *partlist* can be a single number, a range (m-n), a set (m:n), or a combination of ranges and sets separated by colons. Deferred implementation on IRIX systems.
-q *ocblks*	Number of 4096–byte blocks to be allocated per file system partition. Used by Fortran I/O and auxiliary I/O. Deferred implementation on IRIX systems.
-r *setting*	Activate or suppress the passing of the `O_RAW` flag to the `open`(2) system call. *setting* can be either `on` or `off`. Not available on IRIX systems.
-s *ft*	File type. Enter `text`, `cos`, `blocked`, `unblocked`, `u`, `sbin`, `bin`, `bmx`, or `tape` for *ft*. The `bmx` and `tape` options are not available on IRIX systems.
-t	Temporary file.
-u *bufcnt*	Buffer count. Specifies the number of buffers to be allocated for a file. Deferred implementation on IRIX systems.
-w *setting*	Activate or suppress the passing of the `O_WELLFORMED` flag to the `open`(2) system call. Used by Fortran I/O and FFIO. *setting* may be `on` or `off`. Deferred implementation on IRIX systems.
-x *setting*	Activate or suppress the passing of the `O_PARALLEL` flag to the `open`(2) system call. *setting* can be either `on` or `off`. Not available on IRIX systems.
-y *setting*	Produces repeat counts in list-directed output. *setting* can be either `on` or `off`. The default setting is `on`.
-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*. Available only on IRIX systems.

-C *charcon*	Character set conversion information. Enter ascii, ebcdic, or cdc for *charcon*. If you specify the -C option, you must also specify the -F option. Not available on IRIX systems.
-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.
-L *setting*	Activates or suppresses the passing of the O_LDRAW flag to the open(2) system call. Enter either on or off for *setting*. Not available on IRIX systems.
-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. Deferred implementation on IRIX systems.
-P *scope*	Specifies the scope of a Fortran unit and allows specification of private I/O on UNICOS systems. See the assign(1) man page for details about allowed values for *scope*. Deferred implementation on IRIX systems.
-S *setting*	Uses a comma as a separator in list-directed output. Enter either on or off for *setting*. The default setting is on.
-T *setting*	Activates or suppresses truncation after write for sequential Fortran files. Enter either on or off for *setting*.
-U *setting*	Produces a UNICOS 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 on.
-W *setting*	Produces compressed width in list-directed output. Enter either on or off for *setting*. The default setting is on.

-Y *setting* — Skips unmatched namelist groups in a namelist input record. Enter either `on` or `off` for *setting*. The default setting on UNICOS and UNICOS/mk systems is `off`. The default setting on IRIX systems is `on`.

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.

6.1.3 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*])

IRIX systems: 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)
```

6.2 `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.

6.2.1 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`.

To allocate a file on an SSD-resident or memory-resident file system on a UNICOS system, you can use an `assign` command such as the following:

```
assign -a /ssd/myfile u: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.

6.2.2 File structure selection

Fortran I/O uses five different file structures: text structure, unblocked structure, bmx or tape, pure data structure and COS blocked structure on UNICOS and UNICOS/mk systems (on IRIX systems, the F77 blocked structure is used). 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 (the tape/bmx structure is not available on IRIX systems) :

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`

tape/bmx	`assign -F tape`
	`assign -F bmx`
	`assign -s tape`
	`assign -s bmx`

F77 blocked	`assign -F f77`

For more information about file structures, see Chapter 7, page 73.

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
```

6.2.3 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, page 81, 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 unit 1 is to be connected to a large sequential unformatted file with COS blocked structure on UNICOS or UNICOS/mk systems, enter either of the following assign commands to specify a buffer size of 336:

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

  The buffer size for the example was calculated by multiplying tracks-per-cylinder for one type of disk by the track size in sectors of that disk.

- 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
  ```

6.2.4 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
```

6.2.5 File space allocation (deferred implementation on IRIX systems)

File allocation can be specified with the -n, -c, and -p options to the `assign` command. The -n option specifies the amount of disk space to reserve at the time of a Fortran open. The -c and -p options specify the configuration of the allocated space, the -c option specifies contiguous allocation, and the -p option specifies *striping* (the file system partitions where file allocation will be tried) across disk devices.

There is no guarantee that blocks will actually be allocated on the specified partitions. The *partlist* argument can be one integer, a range of integers (m – n), a set of integers (m: n), or a combination of ranges and sets separated by colons. The partition numbers are submitted directly through the `ialloc`(2) system calls. This option achieves file striping on the specified partition.

You cannot specify the -c and -p options without the -n option. The I/O library issues `ialloc` system calls to preallocate file space and to process the -c and -p attributes. The `ialloc` system call requires the -n attribute to determine the amount of file space to reserve.

For example, to specify file allocation on partitions 0 through 2, partition 4, and partitions 6 through 8, contiguous allocation in each partition, and a total of 100 4096-byte blocks of file space preallocated, you would enter the following command:

```
assign -p 0-2:4:6-8 -c -n 100 foo
```

6.2.6 Device allocation (deferred implementation on IRIX systems)

The `assign -F` command has two specifications that alter the device where a file is resident. If you specify -F sds, a file will be SDS-resident; if you specify

-F mr, a file will be memory resident. Because the sds and mr flexible file I/O layers do not define a record-based file structure, they must be nested beneath a file structure layer when record blocking is needed.

Examples of device allocation follow:

- If unit 1 is a sequential unformatted file that is to be SDS-resident, the following Fortran statements connect the unit:

```
CALL ASNUNIT(1,'-F cos,sds.scr.novfl:0:100',IER)
OPEN(1,FORM='UNFORMATTED')
```

  The -F cos specification selects COS blocked structure. The sds.scr.novfl:0:100 specification indicates that the file should be SDS-resident, that it will not be kept when it is time to close, and that it can grow in size to one hundred 4096-byte blocks.

- If unit 2 is a sequential unformatted file that is to be memory resident, the following Fortran statements connect the unit:

```
CALL ASNUNIT (2,'-F cos,mr',IER)
OPEN(2,FORM='UNFORMATTED')
```

  The -F cos,mr specification selects COS blocked structure with memory residency.

For more information about device allocation, see Chapter 9, page 87.

6.2.7 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, and the -u option specifies the number of buffers maintained for the connected file.

To open unit 1 for direct unformatted access and to specify 10 separate regions of the file that will be heavily accessed, use the following assign command:

```
assign -u 10 u:1
```

6.2.8 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. Use `assign -T off` to suppress truncation in applications in which GETPOS(3F) and SETPOS(3F) are used to simulate random access to a file that has sequential I/O.

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 for UNICOS systems.

Access method assign option	Blocked		Unblocked			
	Blocked -s cos	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				8
Formatted direct I/O WRITE(9,20,REC=)		Valid Default	Valid		Valid	min(recl+1, 8) bytes
Unformatted sequential I/O WRITE(9)	Valid Default		Valid	Valid	Valid	48
Unformatted direct I/O WRITE(9,REC=)			Valid	Valid	Valid Default	max(8, recl) blocks
Buffer in/buffer out	Valid Default		Valid	Valid	Valid	48
Control words	Yes	NEWLINE	No	No	No	
Library buffering	Yes	Yes	No	Yes	Yes	
System cached	No	Yes	No†	No††	Varies	
Ldcache	Yes	Yes	Yes	Yes	Yes	
BACKSPACE	Yes	Yes	No	No	No	
Record size	Any	Any	Any	8*n†††	Any	
Default library buffer size	48	8	0	16	8	

†   Cached if not well-formed

††   No guarantee when physical size not 512 words

††† Everything done to bin should be word boundaries and word size

*   In units of 4096 bytes, unless otherwise specified

a10880

Figure 1. Access methods and default buffer sizes (UNICOS systems)

The following figure summarizes the available access methods and the default buffer sizes for IRIX systems.

	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			1

Formatted direct I/O `WRITE(9,20,REC=)`	Invalid	Valid	Valid		Valid Default	65536 bytes
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	65536 bytes
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	

† In units of 4096 bytes, unless otherwise specified

a11335

Figure 2. Access methods and default buffer size (IRIX systems)

6.3 The `assign` environment file

On UNICOS and UNICOS/mk systems, `assign` command information is stored in the `assign` environment file, which is named `$TMPDIR/.assign` by default. To change the location of the current `assign` environment file, assign the desired path name to the `FILENV` environment variable.

On IRIX systems, you must set the `FILENV` environment variable to use the `assign` command. `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.

The format of the `assign` environment file is subject to change with each UNICOS or IRIX release.

6.4 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 11: 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   [7]

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

Five distinct native file structures are used on UNICOS and UNICOS/mk systems: unblocked, pure, text, cos or blocked, and tape or bmx. On IRIX systems, the unblocked, pure, text, and F77 structures are 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`

- `tape` or `bmx` (not available on IRIX systems)

- `text`

- `u`

- `unblocked`

For more information about valid arguments to the `assign -F` command, see Section 6.2.2, page 64. Table 1 summarizes the Fortran access methods and options.

Table 1. Fortran access methods and options

Access and form	assign –s *ft* defaults	assign –s *ft* options
Unformatted sequential BUFFER IN / BUFFER OUT	blocked / cos*	bin sbin u unblocked bmx/tape
Unformatted direct	unblocked	bin sbin u unblocked
Formatted sequential	text	blocked cos sbin/text bmx/tape
Formatted direct on UNICOS systems	text	sbin/text
Formatted direct on IRIX systems	unblocked	u unblocked
Any type of sequential, formatted, unformatted, or buffer I/O to tape	bmx/tape	bmx/tape
* UNICOS systems only		

On IRIX systems, you cannot specify the default for unformatted sequential access with assign –s. You must use assign –F f77.

7.1 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.

7.1.1 `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 created by a Fortran program on a UNICOS or UNICOS/mk system and with the MIPSpro 7 Fortran 90 compiler on IRIX systems contains records that are the same length. The endfile record is recognized in sequential-access files.

7.1.2 `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, page 49, 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.

7.1.3 `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. On UNICOS and UNICOS/mk systems, padding occurs when noncharacter variables follow character variables in an unformatted record.

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.

7.1.4 `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 a sector. 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.

7.2 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. On UNICOS and UNICOS/mk systems, it is also the default file structure for formatted direct 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, page 81).

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.

7.3 COS or blocked file structure

The cos or blocked file structure uses control words to mark the beginning of each sector 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. This file structure is the default structure for files declared as unformatted sequential access on UNICOS and UNICOS/mk systems.

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, page 81.

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

Field	Bits	Description
m	0-3	Type of control word; 0 for BCW
bdf	11	Bad Data flag (modulo 2^{24}).
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 (4)	ubc (6)	tran (1)	bdf (1)	srs (1)	unused (19)	pfi (20)	pri (20)	fwi (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.

7.4 Tape/bmx file structure (not available on IRIX systems)

The tape or bmx file structure is used for online tape access through the UNICOS tape subsystem. You can use any type of sequential, formatted, unformatted, or buffer I/O to read or write an online tape if this file structure was specified.

Each read or write request results in the processing of one tape block.

This file structure is the default option for doing any type of Fortran I/O to an online tape file. The file structure can be specified with one of the following commands:

```
assign -s bmx
assign -s tape
assign -F bmx
assign -F tape
```

These `assign`(1) commands result in the same file structure. Each read or write request results in the processing of one tape block. This structure can be used only with online IBM-compatible tape files or with ER90 volumes mounted in `blocked` mode. See the Cray document, *UNICOS Tape Subsystem User's Guide*, publication SG–2051, for more information on library interfaces to ER90 volumes.

7.4.1 Library buffers

When using Fortran I/O or FFIO for online tapes and the tape or bmx file structure, all of the user's data passes through a library buffer. The size and number of buffers can affect performance. Each of the library's buffers must be a multiple of the maximum block size (MBS) on the tape, as specified by the `tpmnt -b` command.

On IOS model D systems, one tape buffer is allocated by default. The buffer size is either MBS or (MBS $\times$ *n*), whichever is larger (*n* is the largest integer such that MBS $\times$ *n* ≤ 65536).

On IOS model E systems, the default is to allocate 2 buffers of $4 \times$ MBS each, with a minimum of 65,536 bytes, provided that the total buffer size does not exceed a threshold defined within the library. If the MBS is too large to accommodate this formula, the size of the buffers is adjusted downward, and the number is adjusted downward to remain under the threshold.

In all cases, at least one buffer of at least the MBS in bytes is allocated.

During a write request, the library copies the user's data to its buffer. Each of the user's records must be placed on a 4096-byte boundary within the library buffer. After a user's record is copied to the library buffer, the library checks the remaining free buffer space. If it is less than the maximum block size specified with the `tpmnt -b` command, the library issues an asynchronous write (`writea`(2)) system call. If the user requests that a tape mark be written, this also causes the library to issue a `writea` system call.

When using Fortran I/O or FFIO to read online tapes, the system determines how much data can be placed in the user's buffers. Reading a user's tape mark stops all outstanding asynchronous I/O to that file.

Buffering [8]

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

8.1 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. For example, a 42-sector buffer, when read or written, transfers a track of data between the buffer and the DD-49 disk; a track is a very efficient transfer size.

- 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.

Buffers are used extensively on UNICOS and UNICOS/mk systems. Some of the disk controllers have built-in buffers. The kernel has a cache of buffers called the *system cache* that it uses for various I/O functions on a system-wide basis. The Cray IOS uses buffers to enhance I/O performance. The UNICOS logical device cache (`ldcache`) is a buffering scheme that uses a part of the solid-state storage device (SSD) or buffer memory resident (BMR) in the IOS as a large buffer that is associated with a particular file system. The library routines also use buffers.

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* refers to I/O requests that meet the criteria for UNICOS systems; a well-formed request for a disk file 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.

8.2 Types of buffering

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

8.2.1 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 and the UNICOS system cache 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.

8.2.2 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`

`-s tape/bmx` (deferred implementation on IRIX systems)

`-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. For example, the `-F blankx,cos` option specifies two library buffers for a read of a blank compressed COS blocked file. One buffer handles the blocking and deblocking

associated with the COS blocked control words and the second buffer is used as a work area to process the blank compression. In other cases (for example, -F system), no library buffering occurs.

8.2.3 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.

On UNICOS and UNICOS/mk systems, if requests are well-formed, and the O_RAW flag is set by the libraries when the file is opened, the system cache is bypassed, and I/O is done directly between the user's memory space and the logical device.

On UNICOS systems, if the requests are not well formed, the system cache is used even if the O_RAW flag was selected at open time.

If UNICOS ldcache is present, and the request is well formed, I/O is done directly between the user's memory and ldcache even if the O_RAW bit was not selected.

The following assign(1) command options do not set the O_RAW bit, and it can be expected to use the system cache:

-s sbin

-F *spec* (FFIO, depends on *spec*)

The following assign command options set the O_RAW flag and bypass the system cache on UNICOS and UNICOS/mk systems:

-r on

-s unblocked

-s cos (or -s blocked)

-s bin

-s u

-F *spec* (FFIO, depends on *spec*)

See the *UNICOS Tape Subsystem User's Guide*, publication SG–2051 for details about the use of system caching and tapes.

For the `assign -s cos`, `assign -s bin`, and `assign -s bmx` 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.

The `assign -l` *buflev* option controls kernel buffering. It is used by Fortran I/O, auxiliary I/O, and FFIO. The *buflev* argument can be any of the following values:

- `none`: sets `O_RAW` and `O_LDRAW`

- `ldcache`: sets `O_RAW`, clears `O_LDRAW`

- `full`: clears `O_RAW` and `O_LDRAW`

If this option is not set, the level of system buffering is dependent on the type of open operation being performed.

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

8.2.4 Logical cache buffering

On UNICOS systems, the following elements are part of the *logical device*: `ldcache`, IOS models B, C, and D, IOS buffer memory, and cache in the disk controllers. These buffers are connected to the file system on which the file resides.

8.2.5 Default buffer sizes

The Fortran I/O library automatically chooses appropriate default buffer sizes. On UNICOS systems, you can specify the default buffer sizes for the various types of I/O, using the loader for your compiler. See your loader documentation for complete details.

Devices [9]

This chapter describes the type of storage devices available on UNICOS and UNICOS/mk systems including tapes, solid-state storage device (SSD), disks, and main memory. The type of I/O device used affects the I/O transfer rate.

The information in this chapter is pertinent for UNICOS and UNICOS/mk systems only.

9.1 Tape

The UNICOS tape subsystem runs on all UNICOS systems and is designed for system users who have large-scale data handling needs. Users can read or write to a tape with formatted or unformatted sequential Fortran I/O statements, buffer I/O, and the READDC(3F), READP(3F), WRITEC(3F), and WRITEP(3F) I/O routines.

A Fortran program interfaces with the tape subsystem through the Fortran I/O statements and the I/O library. The *UNICOS Tape Subsystem User's Guide*, publication SG–2051, describes the tape subsystem in detail.

9.1.1 Tape I/O interfaces

There are two different types of tape I/O interfaces: the traditional read[a] and write[a] system calls and *tapelist I/O*, which is unique to magnetic tape processing on UNICOS and UNICOS/mk systems.

Tapelist I/O allows the user to make several I/O requests in one system exchange. It also allows processing of user tape marks, bad tape data, and end-of-volume (EOV) processing.

The system libraries provide the following four common ways to perform tape I/O:

- Through the use of the system calls.

- Through the stdio library, which is commonly used from C. This method provides no means to detect or regulate the positioning of tape block breaks on the tape.

- Through Fortran I/O (not fully supported on UNICOS/mk systems). This provides bad data handling, foreign data conversion, EOV processing, and

high-performance asynchronous buffering. Only a subset of these functions are currently supported through Fortran I/O for the ER90 tape device.

- Through the Flexible File I/O (FFIO) system (not available on UNICOS/mk systems). FFIO is used by Fortran I/O and is also available to C users. It provides bad data handling, foreign data conversion, EOV processing, and asynchronous buffering. FFIO uses tapelist I/O. For more information about FFIO see the INTRO_FFIO(3F) man page. Only a subset of these functions are currently supported through Fortran I/O for the ER90 tape device.

9.1.2 Tape subsystem capabilities

The tape subsystem provides the following capabilities:

- Label processing

- Reading and writing of tape marks

- Tape positioning

- Automatic volume recognition (AVR)

- Multivolume tape files

- Multifile volume allocation

- Foreign dataset conversion on UNICOS and UNICOS/mk systems

- User end-of-volume (EOV) processing

- Concatenated tape files

The tape subsystem supports the following user commands on UNICOS and UNICOS/mk systems:

Command	Description
rls(1)	Releases reserved tape resources
rsv(1)	Reserves tape resources
tpmnt(1)	Requests a tape mount for a tape file
tprst(1)	Displays reserved tape status for the current session ID

tpstat(1)   Displays current tape status

See the *UNICOS Tape Subsystem User's Guide*, publication SG–2051, for more details about the tape subsystem.

9.2 SSD

The SSD is a high-performance device that is used for temporary storage. It is configured as a linear array of 4096-byte blocks. The total number of available blocks depends on the physical size of the SSD.

The data is transferred between the mainframe's central memory and the SSD through special channels. The actual speed of these transfers depends on the SSD and the system configuration. The *SSD Solid-state Storage Device Hardware Reference Manual*, publication HR-0031, describes the SSD.

The SSD has a very fast transfer rate and a large storage capacity. It is ideal for large scratch files, out-of-core solutions, cache space for I/O transfers such as ldcache, and other high-volume, temporary uses.

You can configure the SSD for the following three different types of storage:

- SSD file systems

- Secondary data segments (SDS)

- ldcache

All three implementations can be used within the same SSD. The system administrator allocates a fixed amount of space to each implementation, based on system requirements. The following sections describe these implementations.

9.2.1 SSD file systems

In the UNICOS operating system, file storage space is divided into file systems. A *file system* is a logical device made up of slices from various physical devices. A *slice* is a set of consecutive cylinders or blocks. Each file system is mounted on a directory name so that users can access the file system through the directory name. Thus, if a file system is composed of SSD slices, any file or its descendants that are written into the associated directory will reside on SSD.

To use an SSD file system from a Fortran program, users must ensure that the path name of the file contains the appropriate directory. For example, if an SSD

resident file system is mounted on the /tmp directory, use the assign(1) command to assign a file to that directory and the file will reside on the SSD.

Example:

```
assign -a /tmp/ssdfile u:10
```

Users can also use the OPEN statement in the program to open a file in the directory.

SSD file systems are useful for holding frequently referenced files such as system binary files and object libraries. Some sites use an SSD file system for system swapping space such as /drop or /swapdev. Finally, SSD file systems can be used as a fast temporary scratch space.

9.2.2 Secondary data segments (SDS)

The secondary data segment (SDS) feature allows the I/O routines to treat part of the SSD like an extended or secondary memory. SDS allows I/O requests to move directly between memory and SSD; this provides sustained transfer rates that are faster than that of SSD file systems.

Users must explicitly request SDS space for a process but the space is released automatically when the program ends. Users can request that several files reside in SDS space but the total amount of SDS space requested for the files must be within the SDS allocation limit for the user.

To request SDS space for unit 11 from a Fortran program, use either of the following assign commands:

```
assign -F cos,sds  u:11
```

or

```
assign -F cachea.sds u:11
```

The ssread(2) and sswrite(2) system calls can be called from a Fortran program to move data between a buffer and SDS directly. ssread, sswrite, and ssbreak should not be used in a Fortran program that accesses SDS through the assign command because the libraries use SDSALLOC(3F) to control SDS allocation. Using SSBREAK directly from Fortran conflicts with the SDS management provided by SDSALLOC. The *UNICOS System Calls Reference Manual*, publication SR–2012, describes ssbreak, ssread, and sswrite.

On UNICOS/mk systems, the library does not handle allocation of SDS space from more than one processing element (PE). For files opened from different PEs, do not use SDSALLOC, assign -F sds, or the sds option of assign -F cache or assign -F cachea.

A Fortran programmer can use the CDIR$ AUXILIARY compiler directive to assign SDS space to the arrays specified on the directive line. The name of an auxiliary array or variable must not appear in an I/O statement. See the Fortran Language Reference manuals for your compiler system for a description of this feature. The *UNICOS File Formats and Special Files Reference Manual*, publication SR–2014, describes SDS.

9.2.3 Logical device cache (`ldcache`)

The system administrator can allocate a part of the SDS space as `ldcache`. `ldcache` is a buffer space for the most heavily-used disk file systems. It is assigned one file system at a time. Allocation of the units within each assigned space is done on a **least recently used** basis. When a given file system's portion of the `ldcache` is full, the least recently accessed units are flushed to disk. You do not need to change a Fortran program to make use of `ldcache`. The program or operating system issues physical I/O requests to disk.

9.3 Disk drives

Several permanent mass storage devices or disks are available with UNICOS and UNICOS/mk systems. A disk system for UNICOS and UNICOS/mk systems consists of I/O processors, disk controller units, and disk storage units.

A *sector* is the smallest unit of allocation for a file in the file system. It is also the smallest unit of allocation; all I/O is performed in sectors.

In each disk storage unit, the recording surface available to a read/write head group is called a *disk track*. Each track contains a number of sectors in which data can be recorded and read back. The data in one sector is called a *data block*; the size of the data block varies with the disk type. The number of sectors per track, the number of tracks per cylinder, and the number of cylinders per drive also vary according to the type of disk storage unit. For example, a DD-49 disk storage unit contains 886 cylinders with 8 tracks per cylinder and 42 sectors per track. See the dsk(4) and diskspec(7) man pages for complete details.

The following table lists sector size, track size, and tracks per cylinder for a variety of disks:

Table 2. Disk information

Disk type	Sector size (in words)	Track size (in sectors)	Tracks per cylinder
DD-49	512	42	8
DD-40	512	48	19
DD-41	512	48	15
DD-42	512	48	19
DD-40r	512	48	19
DD-60	2048	23	2
DA-60	8192	23	2
DD-61	512	11	19
DD-62	512	28	9
DA-62	2048	26	9
DD-301	512	25	7
DA-301	2048	25	7
DD-302	4096	28	7
DA-302	16384	28	7

This information is useful when you must determine an efficient buffer size.

Disk-based storage under the UNICOS operating system is divided into logical devices. A logical disk device is a collection of blocks on one or more physical disks or other logical disk devices. These blocks are collected into partitions to be used as file system entities. A block is a sector.

An optional striping capability exists for all disk drives. *Striping* allows a group of physical devices to be treated as one large device with a potential I/O rate of a single device multiplied by the number of devices in the striped group. Striped devices must consist of physical devices that are all of the same type. I/O requests using striping should be in multiples of $n \times ts$ bytes; n is the number of devices in the group and ts is the track size of the disk in bytes (not in words or sectors).

For most disks this figure will be $n \times 4096$ bytes. For DD-60 disks, n must be rounded to the nearest multiple of 4 because its sector size is 16 Kbytes.

Disk striping on some systems can enhance effective transfer rates to and from disks.

9.4 Main memory

The `assign`(1) command provides an option to declare certain files to be memory resident. This option causes these files to reside within the field length of the user's process; its use can result in very fast access times.

To be most effective, this option should be used only with files that will fit within the user's field length limit. A program with a fixed-length heap and memory resident files may deplete memory during execution. Sufficient space for memory resident files may exist but may not exist for other run-time library allocations.

See Chapter 6, page 55, for details about using the `assign` command.

Introduction to FFIO   [10]

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 14, page 183.

10.1 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.
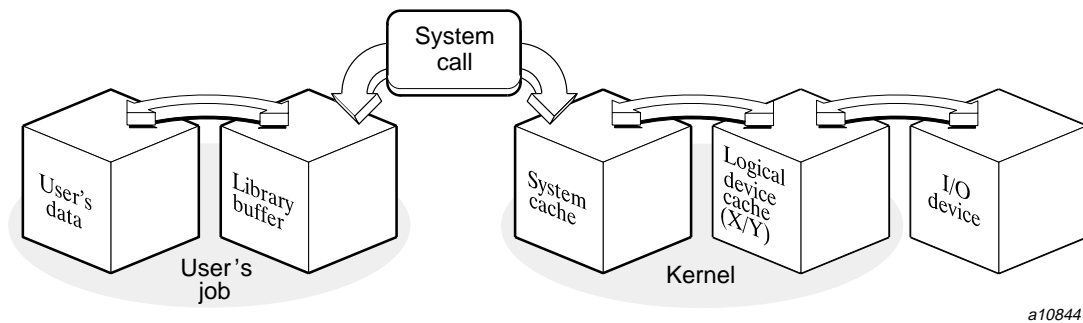
Figure 3. Typical data flow

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 sector 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 3, page 95. 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,blankx,system
```

With this specification, a `WRITE` operation first performs the `user` operation on the data, then performs the `blankx` 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 programmers and C programmers can use the capabilities described in this document. 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

* Word addressable

* Mass Storage (MS) and Direct Random (DR) packages

* Formatted sequential

* Namelist

* List-directed

* Asynchronous queued I/O (AQIO)

The MS package and the DR package includes the OPENMS, WRITMS, READMS, FINDMS, CHECKMS, WAITMS, ASYNCMS, SYNCMS, STINDX, CLOSMS, OPENDR, WRITDR, READDR, and CLOSDR library routines.

10.2 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. For example, unformatted sequential I/O is assigned a default structure of cos on UNICOS systems and UNICOS/mk systems. The -F cos option provides the same structure. The FFIO system

provides detailed control over I/O processing requests. However, to effectively use the `cos` option (or 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 (the default blocking for Fortran unformatted I/O on UNICOS and UNICOS/mk systems), you can specify the following:

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

If you want your file to be F77 blocked (the default blocking for Fortran unformatted I/O on IRIX systems), 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.

A COS blocked blank-compressed file can also be read. The following are the processing steps necessary to do this:

1. Issue system calls to read data from the device.

2. Deblock the data and deliver blank-compressed characters.

3. Decompress the characters and deliver them to the user.

In this case, the *spec* with `system` is on the right end and would be as follows:

```
-F blankx,cos,system
```

You do not need to specify the system *spec* because it is always implied on the right end. To read the COS blocked blank-compressed file, use the following specification:

```
assign -F blankx,cos
```

Because the system *spec* is assumed, it is never required.

10.2.1 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 tape layer uses an array of more sophisticated system calls to handle magnetic tape I/O. The blankx layer passes all data requests to the next lower layer, but it transforms the data before it is passed. The mr layer tries to hold an entire file in a buffer that can change size as the size of the file changes; it also limits actual I/O to lower layers so that I/O occurs only at open, close, and overflow.

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

Table 3. I/O Layers available on all hardware platforms

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
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
user	User-written layer

Table 4. Deferred implementation for IRIX systems

Layer	Function
event	Monitors I/O layers
global	Distributed cache layer
ibm	IBM file formats
mr	Memory-resident file handlers
tape or bmx	UNICOS online tape handling
vms	VAX/VMS file formats

Table 5. Unavailable on IRIX systems

Layer	Function
blankx or blx	Blank compression or expansion layer
c205/eta	CDC CYBER 205/ETA record formats
cdc	CDC 60-bit NOS/SCOPE file formats
er90	ER90 handlers
nosve	CDC NOS/VE file formats
sds	SDS-resident file handlers

10.2.2 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.

The following options all have the same effect. They all specify the sds layer on UNICOS systems and set the initial SDS allocation to 100 512-word sectors:

```
-F sds:100
-F sds.:100
-F sds..:100
```

The following option contains one *spec* for an sds layer that has an *opt* field of scr (which requests scratch file behavior):

```
-F sds.scr
```

The following option requests two *class* es with no *opt* s:

```
-F cos,sds
```

The following option contains two *specs* and requests two layers: cos and sds. The cos layer has no options; the sds layer has options scr and ovfl, which specify that the file is a scratch file that is allowed to overflow, and that the maximum SDS allocation is 1000 sectors:

```
-F cos,sds.scr.ovfl::1000
```

When possible, the default settings of the layers are set so that optional fields are seldom needed.

10.3 Setting FFIO library parameters (UNICOS systems only)

The UNICOS operating system supports a number of library parameters that can be tuned. Sites can use these parameters to change both the performance of the libraries and some of their limits. Through a similar technique, users can also change these parameters when linking an application.

When SEGLDR is invoked, one of its first actions is to read the /lib/segdirs file, which defines the parameters of SEGLDR; this file contains an LINCLUDE directive for the file /usr/lib/segdirs/def_lib, which by default is empty. An administrator can place directives in this file to modify the SEGLDR behavior.

The following HARDREF directives select optional capabilities of the FFIO package to include in the standard libraries compiled into user programs by default.

Table 6. HARDREF Directives

HARDREF =	FFIO option
_f_ffvect	F-type records, fixed length
_v_ffvect	V-type records, variable length
_x_ffvect	X-type records
_cos_ffvect	COS-type records, COS blocking
_tape_ffvect	Magnetic tape handlers
_cdc_ffvect	CDC 60-bit record handlers
_sds_ffvect	SDS-resident file handlers
_mr_ffvect	Memory-resident file handlers
_trc_ffvect	Trace layer
_txt_ffvect	Text-type records, newline separated records
_fd_ffvect	Specified file descriptor
_blx_ffvect	Blank compression handlers

`_cch_ffvect`   Cache layer

Each of these directives refers to a list of function pointers. Each function-pointer list represents the set of routines necessary to process one or more options on the `assign`(1) and/or `asgcmd`(1) commands. Some of these layers are tied to specific hardware, such as tape or SDS. Others are foreign conversion options such as ETA System V-format data. Not all of these layers are loaded into user programs by default. As delivered, the UNICOS operating system can read and write data in many different ways, however, only a subset of these capabilities is loaded into user programs by default, so that user executables are smaller.

If UNICOS source code is available, it is better to change the switches in `fdcconfig.h`, rather than to use these `HARDREF` directives, primarily because `assign` and `asgcmd` still issue warnings to users who use layers disabled in `fdcconfig.h`. Also, changing `fdcconfig.h` is the only way to disable layers that are shipped enabled by default.

Using FFIO [11]

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.

11.1 FFIO on IRIX systems

The FFIO library on IRIX systems 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 on your IRIX system 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. The `system` and `syscall` layers may also issue asynchronous I/O calls if called via the `ffreada`(3C), `ffwritea`(3C), or `fflistio`(3C) routines (deferred implementation on IRIX systems).

11.2 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.

11.2.1 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 and SDS 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.

On UNICOS and UNICOS/mk systems, the `text` layer is also useful in conjunction with the `fdcp`(1) command. The `text` layer provides a standard output format. Many forms of data that are not considered foreign are

sometimes encountered in a heterogeneous computing environment. If a record format can be described with an FFIO specification, it can usually be converted to text format by using the following script:

```
OTHERSPEC=$1
INFILE=$2
OUTFILE=$3
assign -F ${OTHERSPEC} ${INFILE}
assign -F text ${OUTFILE}
fdcp ${INFILE} ${OUTFILE}
```

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

11.2.2 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 several layers designed specifically to handle this binary stream of data. These layers are `syscall`, `sds`, and `mr`. These layers behave the same from the user's perspective; they only use different system resources. 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.

11.2.3 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`, `cachea`, (all available

on UNICOS and UNICOS/mk systems and IRIX systems) and `sds`, and `mr` (available only on UNICOS and UNICOS/mk systems). 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.

11.2.4 Reading and writing COS blocked files

The COS blocking format is the default file structure for all Fortran sequential unformatted files on UNICOS and UNICOS/mk systems, except tape files. The `cos` layer is provided to handle these 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, such as `sds`, you may have to specify a `cos` layer to get COS blocking.

11.3 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 13, page 155.

11.3.1 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 (sectors).

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.

11.3.2 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
sds	SDS-resident I/O (not available on IRIX systems)
cache	Memory-resident buffer cache
mr	Memory-resident (MR) I/O (deferred implementation on IRIX systems)

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.

11.3.3 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 default on UNICOS systems and UNICOS/mk systems depends on the `st_oblksize` field returned from a `stat`(2) system call of the file; if this return value is 0, the default is 489 for ER90 files and 8 for all other files. The *nbufs* argument specifies the number of buffers to use.

11.3.4 The `sds` layer (available only on UNICOS systems)

The `sds` layer is not available on UNICOS/mk systems or on IRIX systems. It is only available on UNICOS systems.

The `sds` layer lets you use the secondary data segment (SDS) feature as an I/O device for almost any file. SDS is one use of the solid-state storage device (SSD). SDS as a device is described in the *UNICOS File Formats and Special Files Reference Manual*, publication SR–2014. If SDS is available, the `sds` layer can yield very high performance. The `sds` transfer rate can approach 2 Gbit/s.

Used in combination with the other layers, COS blocked files, text files, and direct-access files can reside in SDS without recoding. This can provide excellent performance for any file or part of a file that can reside in SDS.

The `sds` layer offers the capability to declare a file to be SDS resident. It features both scratch and save mode, and it performs overflow to the next lower layer (usually disk) automatically. You can declare that a file should reside in SDS to the extent possible. The simplest specification is `assign -F sds fort.1`.

This specification assumes default values for all options on the `sds` layer. By default, the `sds` layer is in `save` mode, which makes the SDS appear like an ordinary file. Because `save` is the assumed mode, any existing file is loaded into SDS when the file is opened. When the file is closed, the data is written back to the disk if the data was changed.

The `sds` layer overflows if necessary. Data that does not fit in the SDS space overflows to the next lower-level layer. This happens regardless of the reason for insufficient SDS space. For example, if you are not validated to use SDS, all of the files that are declared to be SDS-resident immediately overflow. In the previous `assign`(1) example, the overflow goes to disk file `fort.1`. The part of the file that fits in SDS remains there until the file is closed, but the overflowed portion resides on disk.

The `assign -F` command specifies the entire set of processing steps that are performed when I/O is requested. You can use other layers in conjunction with the `sds` layer to produce the desired file structures.

In the previous example, no specification exists for blocking on the file. Therefore, the resulting file structure is identical to the following:

```
assign -s u fort.1
```

This is also identical to the following:

```
assign -F syscall fort.1
```

If a file is COS blocked, a specification must be used that handles block and record control words. The following three examples produce identical files:

```
assign -s cos fort.1
assign -F cos fort.1
assign -F cos,sds fort.1
```

If the file is read or written more than once, adding `sds` to the `assign` command provides speed.

If SDS space is unlimited, almost any unformatted sequential file referenced from Fortran I/O can be declared by using the following command:

```
assign -F cos,sds unf_seq
```

Any formatted sequential file could be declared by using the following command:

```
assign -F text,sds fmt_seq
```

Record blocking is not required for unformatted direct-access files; therefore, any unformatted direct-access file can be declared by using the following command:

```
assign -F sds fort.1
```

In many cases, the `cos` specification is not necessary, but that decision must be made based on the specifics of the particular file and program.

All SDS space that the `sds` layer uses is obtained from the `sdsalloc`(3) library routines. Parameters, environment variables, and rules that pertain to these routines are fully applicable to this I/O technique.

For information about possible fragmentation with SDS, see the `ldcache`(8) man page.

Section 11.4, page 114, contains several `sds` layer examples.

11.3.5 The `mr` layer (deferred implementation on IRIX systems)

The `mr` layer lets you use main memory as an I/O device for many files. Used in combination with the other layers, COS blocked files, text files, and direct-access files can all reside in memory without recoding. This can result in excellent performance for any file or part of a file that can reside in memory.

If the file is small enough to fit in memory and is traversed many times, the wall-clock time can be reduced dramatically by using the `mr` layer to keep the file entirely in memory.

The `mr` layer lets you declare that a file is memory resident. It features both `scratch` and `save` mode, and it performs overflow to the next lower layer (usually disk) automatically.

Memory-resident files can run either in interactive or batch mode. The format for the `mr` layer on the `assign`(1) command is as follows:

```
assign -F mr.savscr.ovfopt:min:max:incr
```

The `assign -F` command specifies the entire set of processing steps that are performed when I/O is requested. If the `mr` layer is specified alone, the resulting file structure is identical to the following:

```
assign -s unblocked fort.1
```

If a file is COS blocked, you must specify the handling of block and record control words as in the following example:

```
assign -s cos fort.1
```

The previous `assign` specification is identical to both of the following:

```
assign -F cos fort.1
assign -F cos,mr fort.1
```

Section 11.4, page 114, contains several `mr` program examples.

11.3.6 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 sector 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/HUMCHUNKS)
read 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 = 40000 bytes = 97 blocks
```

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

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

11.4 Sample programs for UNICOS systems

The following examples contain coding examples using the different layers that
were discussed previously.

Example 12: `sds` using buffer I/O

The following is an example of a batch request shell script that uses an `sds`
layer with buffer I/O. In the following example, a batch job named `exam1`
contains the following statements:

```
#QSUB -r exam1 -lT 10 -lQ 500000
#QSUB -eo -o exam1.out
set -x
cd $TMPDIR
cat > ex1.f <<EOF
      program example1
      double precision r(512),r1(512)
      open(1,form='unformatted')
      do 100 k=1,100
        do 10 j=1,512
10      r(j)=j+k
        buffer out(1,1)(r(1),r(512))
        if(unit(1).ne.-1.0)then
          print *,"error on 1 bufferout=",unit(1)," rec=",k
        end if
100   continue
      rewind 1
      do 200 k=1,100
        bufferin(1,1)(r1(1),r1(512))
        if(unit(1).ne.-1.0)then
          print *,"error on bufferin 1=",unit(1)," rec=",k
        end if
200   continue
      close(1)
```

```
          end
EOF
f90  ex1.f -o ex1              # compile and load
assign -R                      # reset assign parameters
assign -F sds f:fort.1         # assign fort.1 to SDS
./ex1                          # execute
```

If `fort.1` does not exist, the example could use a blocked or unblocked file
structure for the `fort.1` file. If COS record blocking is preferred, the last
`assign` command should be changed to add the following `cos` layer:

```
assign -F cos,sds fort.1
```

The `-lQ 500000` parameter on line 1 of the example indicates to NQS that the
job needs 500,000 bytes of SDS for execution. It is rounded up to the nearest
sector. The size was calculated from the formula ($512 \times 100 \times 8 + numctl$);
numctl is the number of control words needed for the number of records and
sectors in the file.

This example assumes that `save` and `overflow` are the default options for the
sds layer. File `fort.1` is saved in the current directory. If the data exceeds the
maximum available space, the file overflows to disk. As shown, the example
does not exceed the maximum available space.

Example 13: Unformatted sequential `sds` example

In the following batch job, `ex3`, the program performs unformatted sequential
I/O to a file by using the `cos` and `sds` layers:

```
#QSUB -r ex3 -lT 10 -lQ 500000
#QSUB -eo -o exam3.out
date
set -x
cd $TMPDIR
cat > test.f <<EOF
      program example3
      integer r(512),r1(512)
      data r/512*0.0/
      open(1,form='unformatted')
      do 100 k=1,100
        write(1)r
100   continue
      rewind(1)
      read(1)r1
```

```
               endfile(1)
               backspace(1)
               backspace(1)
               read(1)r1
               close(1)
               end
EOF
f90 -o test test.f      # compile and load
assign -R               # reset assign parameters
assign -F cos,sds.scr.novfl fort.1
                        # assign file fort.1 to be COS blocked, and
                        # reside in SDS space with scr scratch
                        # novfl overflow option enabled
./test                  # execute
```

In this example, the program uses BACKSPACE statements; therefore, the specified file must be a COS blocked file structure. Record delimiters are needed for repositioning through a backspace. If the program did not use BACKSPACE statements, the file could be assigned an unblocked structure by using the following command:

```
assign -F sds.scr.novfl fort.1
```

The scr parameter directs the library not to save the file on disk when unit 1 is closed. The novfl parameter in this example specifies that the program aborts if the amount of data written exceeds the maximum available space.

Example 14: sds and mr with WAIO

The following batch job, exam5, uses the word-addressable (WA) package with an sds and mr layer. The arguments to the assign(1) command in this example are as follows:

```
-F mr.scr.ovfl::25:,sds.scr.ovfl::50: fort.1
```

The arguments specify that the first 25 blocks of the file reside in main memory. If the program writes more than 25 blocks, the overflow is written to the next lower layer (sds in this example). If the program writes more than 75 blocks, the remainder or overflow is written to the next lower layer (disk). No space is reserved on disk for the amount of blocks that are memory-resident or SDS-resident because nothing will be saved. The file is deleted when the file is closed. If the save option is specified as in the following example, the first 25 blocks of the written file will be memory resident:

```
-F mr.save.ovfl::25:,sds.save.ovfl::50: fort.1
```

If the program writes another 50 blocks to the file, 25 blocks are skipped in the next lower layer (`sds`) and the 50 blocks are written after skipping the first 25 blocks. If the program writes more than 75 blocks to the file, the first 75 blocks of the next lower layer (disk) are skipped and the write starts at the 76th block of the disk file. When the file is closed, the `mr` and `sds` layers are written to the file.

```
#QSUB -r exam5 -lT 10 -lQ 500000
#QSUB -eo -o ex5.out
date
set -x
cd $TMPDIR
cat > ex5.f <<EOF
      program example5
      dimension r(512),r1(512)
      iblks=10                      !use a 10 block buffer
      istats=1                      !print out I/O stats
      call wopen(1,iblks,istats,ier)
      if(ier.ne.0)then
         print *,"error on wopen=",ier
         goto300
      end if
      iaddr=1
      do 100 k=1,100
        do 10 j=1,512
10      r(j)=j+k
        call putwa(1,r,iaddr,512,ier)
        if(ier.ne.0)then
           print *,"error on putwa=",ier," rec=",k
           goto300
        end if
        iaddr=iaddr+512
100    continue
      iaddr=1
      do 200 k=1,100
        call getwa(1,r1,iaddr,512,ier)
        if(ier.ne.0)then
           print *,"error on getwa=",ier," rec=",k
           goto300
        end if
        iaddr=iaddr+512
200    continue
300    continue
```

```
      call wclose(1)
      end
EOF
f90  ex5.f -o ex5      # compile and load
assign -R              # reset assign parameters
assign -F mr.scr.ovfl::25:,sds.scr.ovfl::50 f:fort.1
                       # assign mr layer with 25 block limit &
                       # sds layer with 50 block limit to fort.1
./ex5                  # execute
```

Example 15: Unformatted direct `sds` and `mr` example

In the following example, batch job `ex8` contains a program that uses
unformatted direct-access I/O with both an `sds` and an `mr` layer:

```
#QSUB -r ex8 -lT 10 -lQ 500000
#QSUB -eo -o ex8.out
date
set -x
cd $TMPDIR
cat > ex8.f <<EOF
      program example8
      dimension r(512)
      data r/512*2.0/
      open(1,form='unformatted',access='direct',recl=4096)
      do 100 i=1,100
        write(1,rec=i,iostat=ier)r
        if(ier.ne.0)then
           if(ier.eq.5034)then
              print *,"overflow to disk at record=",i
           else
              print *,"error on write=",ier
           end if
        end if
100   continue
      do 200 i=100,1,-1
        read(1,rec=i,iostat=ier)r
        if(ier.ne.0)then
           print *,"error on read=",ier
        end if
200   continue
      close(1)
      end
EOF
```

```
f90  ex8.f -o ex8            # compile and compile
assign -R                    # reset assign parameters
assign -F mr.scr.ovfl::50:,sds.scr.ovfl::100: fort.1
                             # assign file fort.1 to be mr with a
                             # 50 block limit, then write the next
                             # 100 blocks to sds.
./ex8                        # execute
```

The program writes the first 50 blocks of fort.1 to the memory-resident layer.
The next 50 blocks overflow the mr buffer and will be written to the sds layer.
No space is reserved in SDS for the first 50 blocks of the file. If the program
writes more than 150 blocks to fort.1, the overflow is written to the next
lower layer (disk). Because the scr option is specified, the file is not saved
when fort.1 is closed.

Example 16: `sds` with MS package example

In this example, batch job ex9 contains a program that uses the MS
record-addressable package and an sds layer:

```
#QSUB -r ex9 -lT 10 -lQ 500000
#QSUB -eo -o ex9.out
set -x
cd $TMPDIR
cat > ex9.f <<EOF
      program example9
      dimension r(512)
      dimension idx(512)
      data r/512*2.0/
      irflag=0
      call openms(1,idx,100,0,ier)
      do 100 i=1,100
         call writms(1,r,512,i,irflag,0,ier)
         if(ier.ne.0)then
            print *,"error on writms=",ier
            goto300
         end if
100   continue
      do 200 i=1,100
         call readms(1,r,512,i,irflag,0,ier)
         if(ier.ne.0)then
            print *,"error on readms=",ier
            goto300
```

```
          end if
200    continue
300    continue
       call closms(1,ier)
       end
EOF
f90  ex9.f -o ex9        # compile
assign -R                # reset assign parameters
assign -F sds.scr.novfl:50:100: fort.1
                         # assign file fort.1 to sds minimum 50
                         # block allocation and maximum 100 blocks.
                         # If more than 100 blocks are written,that
                         # portion of data overflows to disk.
./ex9                    # execute
```

Because the scr option was specified, the file is not saved when fort.1 is
closed.

Example 17: mr with buffer I/O example

The following program uses a memory-resident layer with buffer I/O:

```
 cat> ex2.f<<EOF
      program example2
      integer r(512),r1(512)
      integer ipos1(100)
      open(1,form='unformatted')
      do 100 k=1,100
      do 10 j=1,512
10    r(j)=j+k

       call getpos(1,1,ipos1(k))
       buffer out(1,1)(r(1),r(512))
       if(unit(1).ne.-1.0)then
         print *,"error on bufferout=",unit(1)," rec=",k
         goto300
       end if
100   continue
      do 200 k=100,1,-2
        call setpos(1,1,ipos1(k))
        buffer in(1,1)(r1(1),r1(512))
        if(unit(1).ne.-1.0)then
          print *,"error on bufferin =",unit(1)," rec=",k
          goto300
```

```
          end if
200  continue
300  continue
     close(1)
     end
EOF
f90 ex2.f -o ex2        # compile and load
assign -R               # reset assign parameters
assign -F mr fort.1     # assign file fort.1 to memory
./ex2
```

You may specify a blocked or unblocked file structure for file `fort.1` if the file does not exist.

Example 18: Unformatted sequential `mr` examples

The following program uses an `mr` layer with unformatted sequential I/O:

```
     program example4a
     integer r(512)
     data r/512*1.0/
C    Reset assign environment, then assign file without FFIO
C    to be read back in by subsequent program.
     call assign('assign -R',ier1)
     call assign('assign -a /tmp/file1 -s unblocked f:fort.1',ier2)
     if(ier1.ne.0.or.ier2.ne.0)then
       print *,"assign error"
       goto200
     end if
     open(1,form='unformatted')
C    write out 100 records to disk file: /tmp/file1
     do 100 k=1,100
       write(1)r
100  continue
     close(1)
200  continue
     end
```

In program unit `example4b` which follows, the assign arguments contain the following options to use blocked file structure:

```
assign -R
assign -a /tmp/file1 -F cos,mr.save.ovfl u:3
```

example4b writes an unblocked file disk file, /tmp/file1. If you want to use a blocked file structure, the assign command arguments should contain the following in program unit example4a:

```
assign -R
assign -a /tmp/file1 f:fort.1

      program example4b
      integer r(512)
C     Reset assign environment, then assign file
C     with an mr layer.
      call assign('assign -R',ier1)
      call assign('assign -a /tmp/file1
     &          -F mr.save.ovfl u:3',ier2)
      if(ier1.ne.0.or.ier2.ne.0)then
        print *,"assign error"
        goto300
      end if
C     open the previously written file '/tmp/file1',
C     load it into memory
      open(3,form='unformatted')
C     read 5 records
      do 200 k=1,5
        read(3)r1
200   continue
       rewind(3)
      close(3)
300   continue
      end
```

A sequential formatted file must always have a text specification before the residency layer specification so that the I/O library can determine the end of a record.

Example 19: `mr` and MS package example

The following program uses an mr layer with the MS record-addressable package:

```
      program example10
      dimension r(512)
      dimension idx(512)
      data r/512*2.0/
      irflag=0
```

```
      call assign('assign -R',ier1)
C
C     Assign memory-resident file with overflow, an initial
C     size of 10 512-word blocks, a 200 block limit before
C     overflow and an increment size of 20 blocks.
      call asnunit(1,'-F mr.save.ovfl:10:200:20',ier2)
      if(ier1.ne.0.or.ier2.ne.0)then
        print *,"assign error"
        goto400
      end if
      call opendr(1,idx,100,0,ier)
      do 100 i=1,100
        call writdr(1,r,512,i,irflag,0,ier)
          if(ier.ne.0)then
            print *,"error on writdr=",ier
            goto300
          end if
100   continue
      do 200 i=1,100
        call readdr(1,r,512,i,irflag,0,ier)
        if(ier.ne.0)then
          print *,"error on readdr=",ier
          goto300
        end if
200   continue
300   call closdr(1,ier)
400   continue
      end
```

Foreign File Conversion [12]

This chapter contains information about data conversion, a discussion on 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 Fortran data from Cray PVP systems floating-point representation to floating-point representation for Cray MPP systems and vice versa. Most of the routines discussed in this chapter are **not available** on Cray MPP systems or on IRIX systems. For complete implementation details, see the individual man pages or the `INTRO_CONVERSION`(3F) man page.

12.1 Conversion overview

Data can be transferred between UNICOS systems and other computer systems in several ways. These methods include the use of stations supplied by Cray Research and online tapes and utilities built on TCP/IP (such as `ftp`).

Cray Research supports foreign data conversion to and from IBM, VAX/VMS, CDC NOS/VE, CYBER 205, and the Institute of Electrical and Electronics Engineers (IEEE) format. 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 on UNICOS systems, 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 data to and from the *foreign* data formats. Cray Research provides these routines for many data formats. This is discussed in more detail in Section 12.3.1, page 132.

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 Section 12.3.2, page 133.

12.2 Transferring data

This section describes several ways to transfer data, including using the `fdcp` tool, magnetic tape, and station conversion facilities.

12.2.1 Using `fdcp` to transfer files (not available on IRIX systems)

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.

The following four examples show some uses of `fdcp`:

Example 20: Copy VAX/VMS tape file to disk

Copy a VAX/VMS tape file from labeled tape to disk, converting the logical records to Cray's UNICOS text records. The resulting file is a UNICOS text file that contains the VAX data.

```
assign -F vms.v.tape tapefile
assign -F text diskfile
fdcp tapefile diskfile

vi diskfile          # process the data
```

Example 21: Copy unknown tape type to disk

Copy a tape of unknown type to disk and preserve all tape blocks and tape marks in a COS blocked file. For this example, no `assign` command is necessary for the tape. If an `assign` command was needed, it would be the following:

```
assign -F bmx tapefile
```

The following `assign` and `fdcp` commands are required for disk files:

```
assign -F cos diskfile
fdcp tapefile diskfile
```

After examining the tape label, you discover that it is an IBM tape with fixed-length 240-byte records in 4800-byte tape blocks. Read this disk file by reassigning the file, using the following command:

```
assign -F ibm.fb:240:4800,cos -N ibm diskfile
```

Then execute the program using the following command:

```
./a.out diskfile
```

If this information was available when the tape was mounted, the tape could have been read directly by using the following command:

```
assign -F ibm.fb:240:4800,bmx -N ibm tapefile
```

Then the program could be executed with the following command:

```
./a.out tapefile
```

Example 22: Creating files for other systems

Use `fdcp` to create files for use on other systems. To create a COS blocked blank-compressed file to send to a COS site on tape, do the following:

```
assign -F blankx,cos tapefile
assign -F text diskfile
fdcp diskfile tapefile
```

If this sample tape will contain more than one file, the `assign -F text diskfile` syntax can be replaced by any of the following syntaxes:

```
assign -F text file1
assign -F text file2
assign -F text file3
assign -F text file4

fdcp file1,file2,file3,file4 tapefile
```

This creates a COS transparent tape that contains four files.

Example 23: Copying to UNICOS text files

The same tape created in Example 22 (if created on the COS site) can be read on UNICOS in the same way. The following command copies each of the files on the tape to a separate UNICOS text file.

```
assign -F blankx,cos tapefile
assign -F text file1
assign -F text file2
assign -F text file3
assign -F text file4

fdcp tapefile file1,file2,file3,file4
```

12.2.2 Moving data between systems

This section describes the following ways to move data between the UNICOS machine and other machines:

- Station conversion facilities (not available on CRAY T3E systems or IRIX systems)

- Magnetic tapes (not available on IRIX systems)

- TCP/IP, ftp, and other networks

12.2.2.1 Station conversion facilities

When a station converts a front-end file to a UNICOS file or a UNICOS file to a front-end file, two basic conversion options are available:

- Conversion of the file's internal block and record control structures

- Conversion of character data such as IBM EBCDIC and CDC display code to and from ASCII; conversion of data types other than character is not available with station facilities.

Data conversion is done only with the knowledge of the front-end computer's data type formats. The station software performs character and control structure conversions.

When a front-end file is processed by using fetch, acquire, or dispose commands, the -f *parameter* option to those commands defines if a file is processed in COS blocked format and if character conversion should be

performed. See the fetch(1), acquire(1), and dispose(1) man pages for more details.

The following are some of the options for the format *parameter*:

Option	Description
bb	Binary blocked. The station converts between COS file and front-end file record blocking structures. The data in the resulting file remains unchanged.
tr	Transparent. No blocking, deblocking, or character conversion are performed. The resulting unblocked file is a bit copy of the original file. The exact format of this data depends on the front-end operating system.
cb	Character blocked. The station converts between COS file and front-end file record-blocking structures. The file is assumed to consist of character data which is converted between ASCII and the front-end character set. Cray Research blank compression and expansion also occur.
ud	UNIX operating system data. This is often used to transfer text files, and it is the default between UNIX text file format and front-end record blocking structures.

All of the stations (including IBM MVS and VM, CDC, and VMS) support all of the preceding options. The bb file format conversion differs for each station.

12.2.2.2 Magnetic tape

The simplest way to move data between machines is to carry a magnetic tape from one computer room to another. The following commands are most relevant to this process:

Command	Description
rsv	Reserves tape devices
tpmnt	Mounts the tape
assign	Declares Fortran FFIO processing options
rls	Releases the tape

Many options are available with these commands; only a subset is discussed in this manual. For complete details, see the Cray Research publication, the

UNICOS Tape Subsystem User's Guide, publication SG–2051. When mounting a tape, it is assumed that the `rsv` command was used to reserve a tape drive and the `rls` command will be used to release the tape and the drive when you finish.

The `tpmnt -T` option that allows user tape marks and the `tpmnt -l` option that specifies a label type are relevant to data conversion. For VAX/VMS data conversion, remember that tapes are written in entirely different formats depending on the tape labeling.

When writing a tape, this factor must be considered: if a tape will be read on a foreign system, the tape label should contain information about the record type in each file. The request to set up this information is made by specifying option(s) on the `tpmnt` command. These options do not affect the actual record format of the data in the file; they simply request the tape subsystem to place the information in the appropriate labels.

The following options are available on the `tpmnt` command:

- `-F` *record-format* specifies the value of the record format field in the HDR2, EOV2, and EOF2 labels of the ANSI and standard IBM labels. The *record format* value is used when creating labels. If you do not specify this option when creating a new file, the default record format is `U`. The *record-format* option can have the following values:

`F`	Fixed length (for both ANSI label and IBM standard label). This corresponds to IBM `F` and `FB` formats and VMS `F` format.
`D`	Variable length with zoned decimal length indicator (for ANSI label). This corresponds to VMS `D` and `S` formats and NOS/VE `D` and `S` formats.
`U`	Undefined length (for both ANSI label and IBM standard label). This corresponds to IBM and NOS/VE `U` formats.
`V`	Variable length (for standard IBM label). This is appropriate for all IBM `V`, `VB`, and `VBS` files.

  An optional second character can also be specified. This indicates the attributes of the data and can be one of the following (you must determine if the target system requires these values):

`B`	Blocked records
`S`	Spanned or standard records

R   Blocked and spanned or standard records

- -L *record-length* specifies the maximum number of bytes in a record length. It is used differently on various systems and usually corresponds to the logical record length. If this option is not present and a new file is being created, an installation default value is used.

- -b *block-size* is the same as the mbs parameter in many of the FFIO specifications for foreign record types. It is placed in the label of the tape when a file is created and is often unnecessary (but can be specified if a particular value is needed). It is checked at processing time. If it is small, the tape writing fails. An installation default is used if this option is not present and a new file is being created.

When creating a tape to use on another system, you can place this information in the tape labels. This can be important when reading these tapes on systems other than the UNICOS operating system, where it is unused.

12.2.2.3 TCP/IP and other networks

Several network utilities allow users to move data between computer systems. In this manual, the rcp and ftp utilities are discussed. These utilities work very well transferring files between systems based on UNIX software.

When transferring a file to a foreign system, FFIO can create the file in the correct foreign format but ftp cannot establish the right attributes on the file so that the foreign operating system can handle it correctly. Therefore, ftp is not useful as a transfer agent on IBM and VMS systems for binary data. Its utility is limited to those systems that do not embed record attributes in the system file information.

12.3 Data item conversion

The UNICOS operating system provides both implicit and explicit conversion of data items. Explicit conversion means that the user's code must invoke the routines that convert between UNICOS systems and other vendors' internal representations.

Options to the assign(1) command controls 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 provides direction in choosing a conversion type.

12.3.1 Explicit data item conversion (not available on IRIX systems)

The Cray Research Fortran library contains a set of subroutines that convert between Cray Research data formats and the formats of various vendors. These routines are callable from any programming language supported by Cray Research. The explicit conversion routines convert between IBM, VAX/VMS, CDC, NOS/VE, CYBER 205, or IEEE binary data formats and Cray Research binary data formats. 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. These are the recommended routines, and they replace the older explicit routines. See Appendix A, page 261, for a complete listing of the older, outmoded routines.

Table 7 lists subroutines that convert foreign types to Cray Research types.

Table 7. Foreign file conversion routines

Vendor	Foreign -> Cray	Cray -> foreign
IBM	IBM2CRAY	CRAY2IBM
VAX/VMS	VAX2CRAY	CRAY2VAX
CDC (NOS)	CDC2CRAY	CRAY2CDC
CDC (NOS/VE)	NVE2CRAY	CRAY2NVE
Ultrix	IEU2CRAY	CRAY2IEU
CDC CYBER 205	ETA2CRAY	CRAY2ETA
User conversion	USR2CRAY	CRAY2USR
Site conversion	STE2CRAY	CRAY2STE

Table 8 lists subroutines that convert IEEE formats to Cray Research IEEE and non-IEEE formats.

Table 8. Foreign file conversion routines (IEEE)

Vendor	Foreign -> Cray	Cray -> foreign
Cray (IEEE)	CRI2CRAY (converts Cray IEEE to Cray non-IEEE)	CRY2CRI (converts Cray non-IEEE to Cray IEEE)
	CRI2IBM (converts Cray IEEE to IBM)	IBM2CRI (converts IBM to Cray IEEE)
Cray (non-IEEE)	CRY2CRI (converts Cray to IEEE)	CRI2CRY (converts IEEE to Cray)
IEEE (generic)	IEG2CRAY (converts IEEE generic to Cray non-IEEE)	CRAY2IEG (converts Cray to IEEE generic)
IEEE (generic)	IEG2CRI (converts IEEE generic to Cray IEEE)	CRI2IEG (converts Cray IEEE to IEEE generic)

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

12.3.2 Implicit data item conversion (deferred implementation on IRIX systems)

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 in the following list:

`-N` option	`-C` default
ibm	ebcdic
vms	ascii
cdc	cdc
nosve	ascii
c205	ascii
ieee	ascii
user	ascii
site	ascii
ultrix	ascii
ieee_dp	ascii
ultrix_dp	ascii
ibm_dp	ebcdic
vms_dp	ascii

Cray Research supports the following implicit data conversion:

- 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.

- Conversion of the supported tape/disk record formats only for AQIO or `CALL READ/WRITE`. No data item conversion is performed.

Generally, read, write, and rewind are supported for all record formats. Other capabilities, such as backspace and `GETPOS/SETPOS` are usually not available,

but they can be made to work if a blocking type can be used to support it. See the sections on the specific layers for complete details.

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 between its Cray Research representation and a foreign representation, according to its Fortran data type. Table 9, page 135 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 Cray Research data elements. If the value in a Cray Research 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 Cray Research 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 9. Supported foreign I/O formats and data types

Vendor data type	Record formats	Foreign data types	Cray Research data types

IBM	U, F, FB, V, VB, VBS	`INTEGER*2`
		`INTEGER*4`
		`DOUBLE PRECISION`
		`COMPLEX*4`
		`LOGICAL*4`
		`CHARACTER` (EBCIDIC)
			`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)

Vendor data type	Record formats	Foreign data types	Cray Research data types

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)

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 CF77)	`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)

Vendor data type	Record formats	Foreign data types	Cray Research data types
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 Cray short integer (INTEGER*2) type

Note 2: On Cray MPP systems, Fortran 90 will not implicitly correct INTEGER*2 data. Explicit conversion is supported.

Note 3: Special data conversion types ibm_dp , ieee_dp , ultrix_dp , and vms_dp are available. These types modify the conversion of real data if any of the following conditions apply to all real data items that are written or read to a unit with implicit data item conversion: the I/O list item is of type DOUBLE PRECISION and the -dp compiler option was specified when compiled on the Cray Research system; the I/O list item is of type REAL*8 and the other vendor supports REAL*8 as 8-byte real; or the I/O list item is of type REAL*8 or REAL and the program was compiled on the other (foreign) vendor system with an option which maps REAL*8 or REAL to 8-byte real.

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.

12.3.3 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.

12.3.3.1 Station conversion (not available on IRIX systems)

The following are some of the advantages of using the station software to convert data:

- The system overhead associated with data conversion is placed on the front-end system rather than on the UNICOS system.

- You do not have to change source code.

- Your Cray Research job processes only with Cray Research format data.

Some disadvantages of using the front-end station for conversion include the following:

- Binary data cannot be converted.

- Front-end systems have a relatively slow processing speed.

12.3.3.2 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.

12.3.3.3 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) or asgcmd(1) command (asgcmd is not available on IRIX systems).

- 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.

12.3.4 Disabling conversion types (not available on IRIX systems)

The subroutines required to handle data conversion must be loaded into absolute binary files. By default, the run-time libraries include references to routines required to support the forms of implicit conversion enabled in the foreign data conversion configuration file, usually named <fdcconfig.h>.

If an application requires the use of a conversion routine that is not loaded by default, it can use the loader directives files to activate the routines that support that type of conversion.

It is possible to activate these conversion types for an entire site by using the UNICOS installation tool. Use the following nested menu options:

```
Configure system==>
  SEGLDR loader configuration==>
    Define optional SEGLDR HARDREF directives==>
```

This adds the needed directives in the site-configurable SEGLDR directives file.

12.4 Foreign conversion techniques (not available on IRIX systems)

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

Conversion type	Convert data to/from
CDC 60-bit conversion	CDC CYBER 60-bit machines
COS files	COS systems
CYBER 205 conversion	CDC CYBER 205 and ETA machines
CTSS text files	CTSS format text files
IBM conversion	IBM machines
IEEE conversion	Various types of workstations and different vendors that support IEEE floating-point format
NOS/VE conversion	CDC CYBER machines that run NOS/VE
VAX/VMS conversion	DEC VAX machines that run MVS

12.4.1 CDC CYBER NOS (VE and NOS/BE 60-bit) conversion

Tape formats are physical structures that the operating system superimposes over the user-declared CYBER record manager file structure. The FFIO system supports I-format (internal) and SI -format (system or SCOPE internal) tape formats and files transmitted to the UNICOS operating system from a CYBER disk.

I-format tapes have block sizes that range from 0 to 512 words in exact multiples of 60-bit words. Each block includes a 48-bit block terminator.

SI-format tapes also have block sizes ranging from 0 to 512 words in exact multiples of 60-bit words. Any block smaller than the maximum size (512 words) contains a 48-bit block terminator. This terminator has the same format as that used for I-format tapes.

All CDC sequential tape files are blocked; a block may contain partial records or one or more records. The block structure is intertwined with the physical tape format. Fortran programmers do not use block boundaries. The translation routines construct blocks from records supplied by users and supply users with records as required.

Two CYBER blocking types are supported:

- I (internal): I-blocking contains internal control words (ICWs), which are similar to Cray Research block control words (BCWs). Each block contains an ICW followed by maximum block size in words. The maximum block size (mbs) parameter specifies the maximum number of characters. Records can also span block boundaries. Only W-type records can be used with I-type blocks.

- C (character count): C-blocking implies no blocking. Each block has a fixed number of characters with no special control words internally. The `mbs` parameter specifies the maximum number of characters. You can span records (except CYBER type S) across block boundaries.

A record is the unit of information that is processed on each call for reading or writing. Therefore, the CDC translation routines on UNICOS systems must be aware of the following record types and their analogous structures. Partial record input and output can be achieved using standard buffer I/O requests.

- W-type records are prefixed with a CDC-supplied record control word. The processing of this control word is invisible to users. FFIO routines determine the record length by looking at the control words. This record type is comparable to COS blocked records.

- Z-type records are card-image data. Each record is terminated by a 12-bit byte of zeros in the low-order position of the last 60-bit CYBER word in the record.

- S-type records are system-logical records that contain fixed-size blocks of data terminated by a short block to which is appended a 48-bit level number. The `RS` parameter specifies the maximum number of characters in the record. The record length should be a multiple of ten 6-bit characters. The `assign`(1) command determines conversion characteristics.

Several NOS/VE record formats are supported. One noteworthy restriction is that the `nosve.v` format is not supported on tape.

12.4.2 COS conversions

The COS operating system uses COS blocking for all blocked files. Because the data formats (floating point, character, logical, and integer) are the same as on UNICOS systems with CRI floating point format, and because COS blocking is the default blocking format on the UNICOS operating system, no conversion is necessary when moving unformatted blocked sequential files between the UNICOS operating system on Cray PVP systems with CRI floating point and the COS operating system. Two common file types on COS require some conversion to make them useful on the UNICOS operating system.

The first of these file types is *COS blocked text files.* To handle these, a combination of the `cos` and the `blankx` layers is necessary. The `blankx` layer must process the blank compression that is usually done on COS files. The `cos` layer processes the COS block and record control words that are present in COS text files. To read or write such a file, use the following command:

```
assign -F blankx,cos cosfile
```

With this command, a Fortran program can perform list-directed, formatted, and namelist I/O on `cosfile` as though it were in UNICOS `text` format.

On UNICOS systems, you can also use the `fdcp` command to convert such a file to UNICOS `text` format.

```
assign -F blankx,cos cosfile
assign -F text textfile
fdcp cosfile textfile
```

To create a COS blocked, blank compressed text file, use the following:

```
assign -F blankx,cos cosfile
assign -F text textfile
fdcp textfile cosfile
```

If the COS file contains more than one EOF, specify the `textfile` with `assign -F text.eof`. This directs the `text` layer to use the `~e` marker in the text file to signify an EOF.

The second of these file types is the *COS blocked direct-access file.* Direct-access files on the UNICOS operating system do not contain any blocking information; they are fixed-length record files and rely on the record length for the record boundaries.

On the COS operating system, direct-access files are COS blocked. Unformatted direct-access files are indistinguishable from sequential files. Formatted direct-access files are distinguishable from sequential files only because they are not blank compressed.

Because the FFIO system does not support random positioning on COS blocked files, it is not possible to directly read and write COS direct-access files. You can use `fdcp` to convert the files to a format that can be directly used on the UNICOS operating system. The simplest way to do this is to remove the blocking control words (BCW) from the file. This results in a file that contains all of the fixed-length records in a directly usable format. An example of this follows:

```
assign -F cos cosfile
fdcp cosfile dafile
```

The converse operation requires one more command. You must borrow one of the fixed-length record types because generic fixed-length record type do not exist. An example of this follows:

```
assign -F cosfile
assign -F vms.f.t:800 dafile
fdcp dafile cosfile
```

12.4.3 CDC CYBER 205 and ETA conversion

The CYBER 205 layer is limited to the support of the W-type record. These records are not supported directly on tape; however, if tape files are copied to disk (preferably using fdcp), W-type records can be handled there.

A peculiar feature of the CYBER 205 conversion is the conversion of half-precision floating-point numbers, which are mapped to short integers in the numeric conversion routines. Therefore, if you do this conversion, a short integer array must be an equivalence of a real array to do the I/O.

Example:

```
    integer*2 ioarray(100)
    real fltnum(100)
    equivalence (fltnum,ioarray)
    read(1) ioarray
    do 10 i= 1,100
      call calc(fltnum(i))
C    deal with converted half-precision values
10  continue
    end
```

12.4.4 CTSS conversion

The FFIO system includes two features that are embedded in the blankx and text layers to process CTSS files. CTSS uses its own text file format and uses blank compression on these files. To read and write most CTSS text files, use the following specification:

```
blankx.ctss,text.ctss
```

Because unformatted Fortran files (binary records) are in COS blocked format at CTSS sites running a UNICOS system, conversion is not necessary.

12.4.5 IBM overview

To convert and transfer data between UNICOS systems and an IBM/MVS or VM system, you must understand the differences between the UNICOS file

system and file formats, and those on the IBM system(s). On both VM and MVS, the file system is record oriented.

The most obvious form of data conversion is between the IBM EBCDIC character set and the ASCII character set used on UNICOS systems. Most of the utilities that transfer files to and from the IBM systems automatically convert both the record structures and character set to the UNICOS text format and to ASCII. For example, `ftp` performs these conversions and does not require any further conversion on UNICOS systems.

Binary data, however, is more complicated. You must first find a way to transfer the file and to preserve the record boundaries. If stations are available, this is simple (some examples are shown in the following sections). Few problems are caused by using tapes in transferring the file and preserving record boundaries.

Cray Research supports the following IBM record formats:

Format	Description
U	Undefined record format
F	Fixed-length records, one record per block
FB	Fixed-length, blocked records
V	Variable-length records
VB	Variable-length, blocked records
VBS	Variable-length, blocked, spanned records

For all IBM record formats, the data formats are the same whether you are processing a disk file or processing a tape file.

12.4.5.1 Using the MVS station

To convert IBM foreign data that is transferred using the MVS station, you must perform two basic tasks:

- Convert the data and record formats

- Move the data between the MVS or VM system and UNICOS systems

An example that is common on IBM MVS systems follows. This example starts with a simple Fortran program that creates some data and moves the data to the MVS system. The following shell script shows the details:

```
INTEGER IARR(10)
REAL RARR(15)
```

```
      DO 10 I=1,10
        WRITE(1) IARR,RARR
10    CONTINUE
      STOP
      END

EOF
f90 test.f                 # compile and load program
segldr test.o              # load program
assign -R                  # reset assign
assign -F cos -N ibm fort.1 # select COS blocking (normal default)
                           # and IBM numeric data conversion
./a.out                    # execute
dispose fort.1 -f bb -t 'dsn=array.test,disp=shr'
                           # dispose file to MVS system
```

This program writes 10 records that contain 10 integer values and 15 real numbers. With the IBM numeric conversion enabled, there are ten 32-bit integers in IBM format and fifteen 32-bit IBM real numbers in these records.

During the dispose operation, the key parameter is -f bb. This directs the station to translate COS blocking to IBM blocking according to the file format defined in the file catalog on MVS. The part of this catalog that defines this file structure is called a DCB.

The -t argument specifies the operands of an IBM DD statement. This Job Control Language (JCL) information varies considerably from job to job and user to user.

A Fortran program can read the file that results on the MVS system without special handling.

An example of reading the file back to a UNICOS system follows. This option is usable for all IBM formats.

```
      INTEGER IARR(10)
      REAL RARR(15)
      DO 10 I=1,10
        READ(1) IARR,RARR
 10   CONTINUE
      STOP
      END

EOF
```

```
f90 test.f -o test.o        # compile and load program
assign -R                   # reset assign
assign -F cos -N ibm fort.1 # select cos blocking (normal default)
                            # and IBM numeric data conversion
fetch fort.1 -f BB -t 'dsname=array.test, disp-shr'
                            # fetch file from MVS system, and
                            # convert blocking from what is on
                            # MVS to COS blocked.
./a.out                     # run the program
```

The IBM record formats are not used with the assign command because the station can convert the record format for you. To specify a specific record format, it is more difficult to get the station to transfer the data correctly in both directions without more parameters.

In these examples, the station must interpret the blocking on the IBM/MVS side and translate it into a Cray Research COS blocking format. This is relatively slow and unnecessary because the UNICOS operating system can read the files directly.

The following is an example of a faster IBM MVS station transfer. Use the previous program and try to speed up the station file transfer by using the -f TR option on the fetch and dispose commands. This option causes the station to take the bytes from the IBM disk and to transfer them unchanged and untranslated to the UNICOS system. This produces faster wall-clock times for transferring the data. It costs slightly more in CPU time on a UNICOS system because a UNICOS system must perform the deblocking work previously performed by the station.

The following example assumes that the program that reads the data is compiled and loaded:

```
assign -R                     # reset assign
assign -F ibm.vbs -N ibm fort.1# select IBM VBS record format
                              # and IBM numeric data conversion

fetch fort.1 -f TR -t 'dsn=array.test, disp=shr'
                              # fetch file from MVS system in
                              # 'transparent' mode.
./a.out                       # run the program
```

The example does not try to create the file on a UNICOS system and to send it to MVS in transparent mode; this does not work. You cannot specify proper physical record boundaries in a transparent transfer. The file that results cannot

be used on the MVS system. When fetching and reading files from MVS, however, this is often the best method.

A third option exists to transfer data between the UNICOS operating system and MVS. It requires more knowledge of both the FFIO layers and the IBM disk formats.

The six supported record formats are basic, low-level record types on the MVS operating system. The record format is stored in a part of the file called a DCB. When accessing data, the MVS system invokes appropriate processing to interpret the data in the file so that the user sees only the data, and not the control information that determines logical record boundaries.

Using MVS JCL, a user can specify to the MVS system that the DCB on a given file should be ignored. For example, if you have a VBS format file and you want MVS to read it as though it were a U format file, the system does not interpret the block and segment information embedded in the data. You will see all of the bits in the file on disk, including control words. It is similar to taking a Cray Research blocked file, assigning it as unblocked, and then reading it.

In the same way, when writing a file on the MVS system, you can declare it to be a U format file and the MVS system will not add any control information to your records. This is similar to creating a blocked file on the UNICOS system by inserting your own RCWs and BCWs in an unblocked file. When using the MVS station, the -t option on the fetch or dispose command provides a JCL to define the file and record access methods that MVS uses.

The following example assumes that the program that writes the data is compiled and loaded:

```
assign -R                # reset assign
assign -F ibm.vbs,cos -N ibm fort.1
                         # select IBM VBS record format with blocks
                         # delimited as COS blocked records. Also
                         # specify IBM numeric data conversion
./a.out                  # run the program
dispose fort.1 -f BB -t 'DCB=(RECFM=U),dsname=array.test,disp=shr'
                         # dispose file from MVS system
```

In this example, the assign -F *spec* command includes two layers. The dispose command requests that the COS blocked records on the Cray disk be converted on the MVS side. Using DCB=(RECFM=U) allows the station to write each block as a physical disk block on MVS. This allows the user on the UNICOS system to write the file in whatever format is desired. It also does not rely on the file catalog on the MVS system to determine the format.

An extra step is necessary to change the stored `DCB` on the MVS file so that programs on the MVS side can read the data correctly. Specifically, a `VBS` file was created on a UNICOS system and `dispose -f bb` was used to dispose of it. The station creates and writes the file as a `U` format file through the `TEXT` field on the `dispose`. The station avoids adding control words to the file that are already there. However, you must then correct the `DCB` for the file to match the real format. You can do an empty `dispose` that changes only the `DCB` and leaves the data unchanged. You can also do the empty `dispose` first, then do the `dispose` of the data that tells the station to ignore the `DCB` and to write the file as `U` format.

Other record formats perform like `ibm.vbs`. Any of these can be read or written with any IBM MVS station transfer method. If the faster method is used, you must add the requisite `,cos` to the `-F` specification. A list of common record formats follows:

```
assign -F ibm.v:3280 fort.1    # V format, recsize 3280 bytes
assign -F ibm.vb::16000 fort.1 # VB format,
                               # max block size 16,000 bytes
assign -F ibm.f:1600 fort.1    # F format fixed-length records
                               # of 1600 bytes each.
assign -F ibm.fb:1600:32000 fort.1
                               # Fixed-length records 1600 bytes
                               # each, 20 records per block
assign -F ibm.vbs::1800 fort.1 # VBS format, block size 1800 bytes
```

One different format is the `U` format. This format, unlike all other IBM formats, does not contain any IBM control words to delimit records and lacks a fixed record size. To delimit logical records, the `U` format relies completely on the physical blocks on an IBM disk, or on tape blocks.

These physical records must be translated into some other form to be preserved so that the file will be interpreted correctly on the UNICOS system. This is usually done with a lower-level layer.

12.4.5.2 Data transfer between UNICOS and VM

The primary difference between the VM station and the MVS station is that the record types described here are not known to the VM system, but only to VM applications. The stations are VM applications. Files are stored in a manner similar to that of MVS, in that variable-length blocks exist on the disk. Each of these blocks appears the same as it does on MVS, with block descriptor words and segment descriptor words. The VM system, however, is unaware of the control words stored in the blocks; therefore, you do not have to fool the system

into thinking that the file is in a format that it is not. VM has the same limitations as the MVS station with the usage of TR transfers.

The VM station also handles dispose commands differently than the other stations. The dispose -f command must be set properly to avoid record truncation and unwanted binary data conversion. When disposing directly to CMS minidisk, you must have a cooperating process running on your virtual machine. See the *IBM VM Station Command and Reference for COS*, publication SI-0160.

Two basic methods exist to read and fetch a VB format file on VM/CMS disk from a VM/CMS front end. The examples shown here are abbreviated and do not include all parameters:

- You can fetch the file from the front end by using fetch -f bb. Each disk *block* from the VM system is changed to a COS record. This preserves the control words embedded in the VB data, as in the following:

```
fetch DATA -f BB -t 'dsn=file.name,disp=shr'

assign -F ibm.vb,cos -N ibm DATA
```

- You can fetch the file from VM by using the fetch -f tr (transparent) mode. Use the assign command to declare a file to be read in VB format, and set up the -F specification appropriately, as in the following:

```
fetch TRDATA -f tr -t 'dsname=file.name,disp=shr'

assign -F ibm.vb -N ibm TRDATA
```

12.4.6 Workstation and IEEE conversion

IRIX systems use 32-bit IEEE standard floating point, as do many workstations and personal computers use the IEEE standard floating-point format. 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.

12.4.7 VAX/VMS conversion

Nine record types are supported for VAX/VMS record conversion. This includes a combination of three record types and the three types of storage medium, as defined in the following list:

Record type	Definition
f	Fixed-length records
v	Variable-length records
s	Segmented records

Media	Definition
tr	For transparent access to files
bb	For unlabeled tapes and bb station transfers
tape	For labeled tapes

Segmented records are mainly used by VAX/VMS Fortran. The following are examples of some combinations of segmented records in different types of storage media:

Example	Definition
vms.s.tr	Use as an FFIO specification to read or write a file containing segmented records with transparent access. In the fetch and dispose commands, specify the -f tr option for the file.
vms.s.tape	Use as an FFIO specification to read or write a file containing segmented records on a labeled tape.

`vms.s.bb`	Use as an FFIO specification to read or write a file containing segmented records on an unlabeled tape. In the `fetch` and `dispose` commands, specify the `-f bb` option for the file if it is not a tape.

The VAX/VMS system stores its data as a stream of bytes on various devices. UNICOS systems number their bytes from the most-significant bits to the least-significant bits, while the VAX system numbers the bytes from lowest-significance up. The station and a UNICOS system make this byte-ordering transparent when you use text files. When data conversion is used, byte swapping sometimes must be done.

Character conversion is not necessary for text files transferred using the station because the VAX/VMS system uses the same ASCII character set as UNICOS systems. The station software correctly handles byte-ordering machines for text files.

The process is similar for binary data. You can use the `fetch` and `dispose` commands to move data between the UNICOS system and the VAX. You can move data in several ways.

You can use the `-f bb` transfer format, which places the burden of blocking and unblocking the data to be transferred to the VAX. The VAX station must convert VMS blocking to COS blocking and the opposite is true.

You can also use the `-f tr` transfer option, which results in record formats on UNICOS systems that use the `tr` subfield in the `vms` layer specification.

When using `-f bb` transfers, you must know the format of the file on the VAX side and specify the proper record format when reading/writing the data.

Most VAX/VMS users are aware of only two basic record types, which are `V` and `F` (variable length and fixed length). `F` format on the VAX maps directly to the `vms.f` record types on UNICOS systems.

```
fetch testin -f bb -t 'CRAYMH"USER PASSWD"::SEG.DAT'
assign -a TESTIN -F vms.s.bb::10000,cos -N vms fort.1
assign -a TESTOUT -F vms.s.bb::10000,cos -N vms fort.22
f90 test.f test.o../a.out
dispose TESTOUT -f BB -t 'CRAYMH"USER PASSWD"::[]SEG2.DAT'

    PROGRAM FRED
    REAL
    INTEGER*2 SHORT(50)
C
```

```
C    NOTE THAT BECAUSE -N VMS IS IN USE, THE SHORT INTEGERS
C    BECOME 64 BITS ON THE CRAY RESEARCH SYSTEM
C
10   READ (1,END=99) SHORT, REAL
     CALL PROCESS (STAT,SHORT,REAL)C
C    ...AND ARE CONVERTED BACK ON OUTPUT
C
     WRITE(22) SHORT, REAL
     GOTO 10
99   STOP
     END
     SUBROUTINE PROCESS (STAT,SHORT,REAL)
     REAL REAL(100)
     INTEGER SHORT(50)
C
C    PROCESS THE DATA
C
     RETURN
     END
```

Because the default record type produced on UNICOS systems is v, special work is not required to dispose vms.s.bb or vms.v.bb records. It is also easy to fetch data from the station because the station reads the data properly without user intervention. When using fixed-length record format, you must add the following information to the TEXT field on the dispose command:

```
-t 'CRAYMH"USER PASSWD"::SEG.DAT/RFM=FIX/MRS=256/NORAT'
```

This dispose command works properly for a file created with the vms.f.tr:256 specification.

The RFM, MRS, and NORAT parameters specify a fixed-length record format file, a maximum record size of 256 bytes, and no record attributes.

12.5 Implicit numeric conversions (Cray PVP systems only)

The following HARDREF directives select optional implicit numeric conversions to include in the standard libraries compiled into user programs by default.

Directive	FFIO option
HARDREF=CRAY2IBM	Cray<->IBM implicit numeric conversion
HARDREF=IBM2CRAY	

`HARDREF=CRAY2VAX` `HARDREF=VAX2CRAY`	Cray<->VAX/VMS implicit numeric conversion
`HARDREF=CRAY2NVE` `HARDREF=NVE2CRAY`	Cray<->NOS/VE implicit numeric conversion
`HARDREF=CRAY2IEG` `HARDREF=IEG2CRAY`	Cray<->IEEE implicit numeric conversion
`HARDREF=CRAY2ETA` `HARDREF=ETA2CRAY`	Cray<->ETA implicit numeric conversion
`HARDREF=CRAY2CDC` `HARDREF=CDC2CRAY`	Cray<->CDC 60-bit implicit numeric conversion
`HARDREF=CRI2IBM` `HARDREF=IBM2CRI`	Cray IEEE<->IBM
`HARDREF=CRI2IEG` `HARDREF=IEG2CRI`	Cray IEEE<->Generic IEEE

I/O Optimization   [13]

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. For information about other optimization techniques, see the Cray Research publication, *Optimizing Code on Cray PVP Systems*, publication SG–2192. For information about optimization techniques on UNICOS/mk systems see Section 13.9, page 180. **The remainder of the information in this chapter is used primarily on UNICOS systems, and much of the information is not applicable to IRIX systems**.

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

13.1 Overview

I/O can be represented as a series of *layers* of data movement. Each layer involves some processing. Figure 4, page 156 shows typical output flow from the UNICOS system to disk.
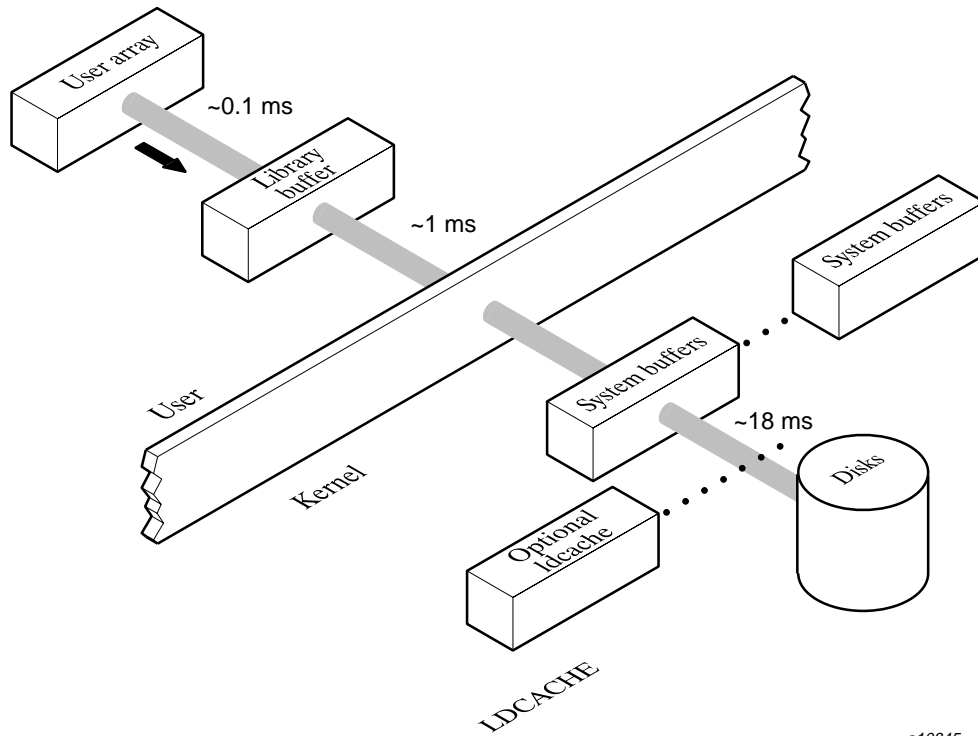
Figure 4. I/O layers

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 4 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.

13.2 An overview of optimization techniques

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

13.2.1 Evaluation tools

Use the following tools to determine the initial I/O performance and to verify improvements in I/O performance after you try different optimization techniques:

- Use the `ja`(1) command to determine whether significant time is spent on I/O in the program (see Section 13.3.1, page 160, for details).

- Use the `procstat`(1) command and `procview`(1) command on the program to help you evaluate the I/O information that the I/O library collects (see Section 13.3.2, page 161, for details).

13.2.2 Optimizations not affecting source code

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

- Use the type of storage devices that are effective for the types of I/O done by the program. Try the `mr` or `ssd` layers (see Section 13.4.1, page 163, or Section 13.4.3, page 167).

- 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 Section 13.4.4, page 168, for details).

- Use file structures without record control information to bypass the overhead associated with records (see Section 13.5.5, page 176, for details).

- Choose file processing with appropriate buffering strategies. The cos, bufa, and cachea FFIO layers implement asynchronous write-behind (see Section 13.5.4, page 175, 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 Section 13.7.1, page 177, for details).

- Determine whether the use of striping, preallocation of the file, and the use of contiguous disk space would improve I/O performance (see Section 13.4.6, page 170, 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 Section 13.5.1, page 171, for details).

Section 11.3, page 108, also provides further information about using FFIO to enhance I/O performance.

13.2.3 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.

13.2.4 Optimizing I/O speed

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

- CPU main memory

- Optional SSD

- 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.

13.3 Determining I/O activity

Before you can optimize I/O, you must first identify the activities that use the most time. The most time-intensive I/O activities are the following:

- System requests

- File structure overhead

- Data conversion

- Data copying

This section describes different commands you can use to examine your programs and determine how much I/O activity is occurring. After you determine the amount of I/O activity, you can then determine the most effective way to optimize the I/O.

The sections that follow make frequent references to the following sample program:

```
      program t
      parameter (nrec=2000, ndim=500)
      dimension a(ndim)
      do 5 i=1,ndim
        a(i) = i
5     continue
      istat = ishell('rm fort.1')
      call timef(t0)
      do 10 i=1,nrec
        write(1) a
10    continue
```

```
c     rewind and read it 3 times
      do 30 i=1,3
        rewind(1)
          do 20 j=1,nrec
            read(1) a
20        continue
30    continue
      call timef(t1)
      nxfer = 8*nrec*ndim*(1+3)
      write(*,*) 'unit 1: ',
+        nxfer/(1000*(t1-t0)),
+        ' Mbytes/sec'
      stop
      end
```

13.3.1 Checking program execution time

The ja(1) command is a job accounting command that can help you determine if optimizing your program will return any significant gain. For complete details about the ja command, see the ja man page.

To use ja(1), enter the following commands:

```
ja
a.out
ja -ct
```

These commands produce the following program execution summary that indicates the time spent in I/O:

```
Command  Started   Elapsed     User CPU   Sys CPU    I/O Wait I/O Wait
  Name      At      Seconds     Seconds    Seconds    Sec Lck  Sec Unlck

======== ======== =========== ========== ========== ======== ==========
a.out    17:15:56    4.5314     0.2599     0.2242     3.9499     0.1711
```

This output indicates that this program has a large amount of I/O wait time. The following section describes how to obtain a profile of the I/O activity in the program.

13.3.2 Generating an I/O profile

A significant part of this example program performs I/O; therefore, you can use `procstat` and related tools to obtain an I/O profile. For complete details about using these tools, see the Cray Research publications, *UNICOS Performance Utilities Reference Manual*, publication SR–2040, and the *UNICOS User Commands Reference Manual*, publication SR–2011, or the `procview`(1) man page.

The `procstat` tool is not available on CRAY T3E systems.

The `procstat` tool set does not require access to the program source files. The run-time library has built-in I/O data collection that is invoked when a program is run with `procstat`. The set of statistics generated usually provides enough information to tune I/O in a Fortran program without altering the source code.

The `procview` tool creates one or more reports from the raw output that the `procstat` command generates. It may also be run interactively, both in line-mode and by using the X Window System interface. The `procview` command presents an interactive menu when no command-line report option is included; otherwise, an output option can be specified and the report output can be redirected to a file.

To run the program under `procstat`, enter the following commands:

```
procstat -R raw a.out
procview -l -Fs raw
```

The `-l` option selects the long form report, and the `-Fs` option selects Fortran files sorted by maximum file size. The resulting report summaries the I/O activity of each Fortran file in the following format:

```
======================================================================

Fortran Unit Number         1
File Name                   fort.1
Command Executed            t1
Date/Time at Open           05/31/91 17:00:19
Date/Time at Close          05/31/91 17:00:26
System File Descriptor      4
Type of I/O                 sequential unformatted
File Structure              COS blocked
File Size                   8032256 (bytes)
Total data transferred      32129024 (bytes)
```

```
Fortran I/O   Count of         Real
Statement    Statements        Time
-----------  ----------  --------------
READ              6000        5.3625
WRITE             2000        1.6484
REWIND               3         .0011
CLOSE                1         .0019


 4014.6    Bytes transferred per Fortran I/O statement
  87.70%   Of Fortran I/O statements did not initiate a system request

System I/O   # of    # Bytes    # Bytes     Wait Time (Clock Periods)
Function    Calls  Processed  Requested      Max       Min      Total
-----------  ------- ---------  --------- --------- --------- ---------
Read            738  24096768   24182784   9010627   135443 865007072
Write           246   8032256    8032256  10674103   133840 253750720
Seek              4       n/a        n/a     42061      3746     55067
Truncate          1       n/a        n/a     17462     17462     17462

System I/O   Avg Bytes  Percent of   Average I/O Rate
Function     Per Call   File Moved   (MegaBytes/Second)
-----------  ---------- ----------  -------------------
Read          32651.4       300.0              4.643
Write         32651.4       100.0              5.276
Seek              n/a         n/a                n/a
Truncate          n/a         n/a                n/a

========================================================================
```

By examining the summary of files examined during a program, you can tell that the following types of files should be optimization targets:

- Files with very high activity rates (total bytes transferred is very large); see the `# Bytes Processed` column in the report.

- Files in which a lot of real time is spent in I/O statements; see the `Real time` and `Total` column figures.

13.4 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 sector 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.

13.4.1 The MR feature

Main memory is extremely fast. Cray Research provides many ways to use memory to avoid delays that are associated with transfers to and from physical devices.

The `mr` FFIO layer, which permits files to reside in main memory, is available on all UNICOS and UNICOS/mk systems. If the memory space is large enough, you can eliminate all system requests for I/O on a file. The previous `procstat` / `procview` report contains the following information:

- The 2000-record file was probably written once and then rewound and read completely three times; this is deduced from the `Count of Statements` on the report.

- The type of I/O was sequential unformatted. The file structure is COS blocked (see `File Structure` on the report).

- Its maximum file size is about 8 Mbytes (see `File Size` on the report).

To apply 8 Mbytes of memory to this file, use the following `assign` command and then rerun the job:

```
assign -F blocked,mr::1961 u:1
```

The maximum size of 1961 is calculated by dividing the file size of 8,032,256 bytes by the sector size of 4096 bytes.

The `-F` option invokes FFIO. The `blocked,mr` specification selects the blocked layer followed by the `mr` layer of FFIO. The `u:1` argument specifies unit 1. Figure 5 shows I/O data movement when you use the `assign` command.
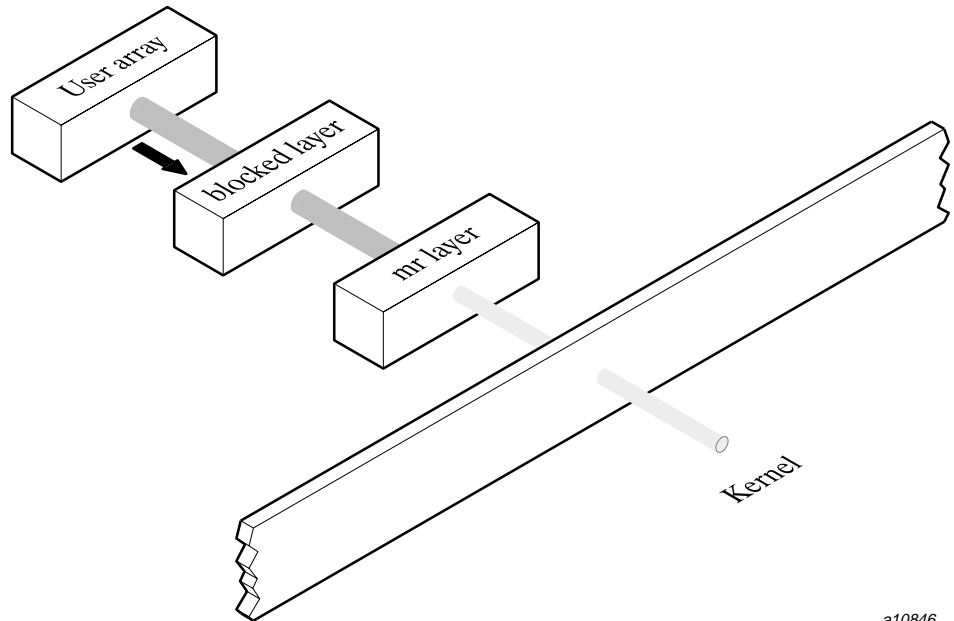

a10846

Figure 5. I/O data movement

The data only moves to and from the buffer of the `mr` layer during the operation of the READ, WRITE, and REWIND I/O statements. It gets moved from disk during OPEN processing if it exists and when SCRATCH is not specified. It gets moved to disk only during CLOSE processing when DELETE is not specified. When the program is rerun under procview, the procview report is as follows:

```
=========================================================================

Fortran Unit Number      1
File Name                fort.1
Command Executed         a.out
Date/Time at Open        09/04/91 17:29:38
Date/Time at Close       09/04/91 17:29:39
System File Descriptor   4
Type of I/O              sequential unformatted
File Structure           COS blocked
File Size                8032256 (bytes)
Total data transferred   8032256 (bytes)
Assign attributes        -F blocked, mr::1961

   Fortran I/O    Count of        Real
   Statement      Statements       Time
  ------------   ----------    --------------
   READ             6000           .1663
   WRITE            2000           .0880
   REWIND              3           .0005
   CLOSE               1           .9055

   1003.7   Bytes transferred per Fortran I/O statement
   99.99%   Of Fortran I/O statements did not initiate a system request

   System I/O  # of      # Bytes      # Bytes    Wait Time   (Clock Periods)
   Function    Calls   Processed    Requested      Max         Min        Total
  -----------  -------  ---------   ---------    ---------   ---------   ---------
   Write          1     8032256      8032256    150197242   150197242   150197242
   Seek           2       n/a          n/a         3655        3654        7309
   Truncate       1       n/a          n/a         5207        5207        5207

   System I/O     Avg Bytes   Percent of     Average I/O Rate
   Function       Per Call    File Moved     (MegaBytes/Second)
  ------------   -----------   ----------    -------------------
     Write        8032256.0      100.0             8.913
     Seek            n/a          n/a               n/a
     Truncate        n/a          n/a               n/a

=========================================================================
```

In the new report, notice the following:

- `Read` time is 0 (no entry for `Read` exists under `System I/O Function`). All of the data that was read was moved from the MR buffer to user space. Data transferred is 0; consequently, the time spent in `Read` is reduced by more than one order of magnitude.

- `Write` time is reduced because the data is moved only to the MR buffer during Fortran `write` s.

- Total write time stays relatively unchanged because the file still has to be flushed to disk at `CLOSE` processing.

13.4.2 Using faster devices

The optional solid-state storage device (SSD) is the fastest I/O device. The SSD stores data in memory chips and operates at speeds about as fast as main memory or 10 to 50 times faster than magnetic disks.

Because SSD capacity is usually much larger than main memory, SSD is used when not enough main memory is available to store all of the possible data.

You can access the SSD through `ldcache`. The system uses SSD to cache the data from file systems that the system administrator selects. Caching is automatic for files in these file systems and their subdirectories.

You can also access the SSD with the FFIO `sds` layer. When this layer is present, library routines use the SSD to hold the file between open and close. You should use the FFIO `sds` layer for files that are larger than the amount of `ldcache` available for the file.

The `SDSLIMIT` and `SDSINCR` environment variables may have significant impact if all subfields are not specified after the SDS keyword (use of these variables is not recommended).

The following timings from a CRAY Y-MP/8 system show the typical effects of optimization on the program used in Section 13.4.1, page 163. In that example, the program writes a file and reads it three times. Because it is unnecessary to save the file afterward, the `.scr` type (scratch file) can be used. See Section 13.5.1, page 171, for more information about scratch files. Some of the following commands appear to produce a range because of the fluctuation in results.

assign command	I/O speed (relative)
Default (no `ldcache`)	1

Default (`ldcache`)	8
(with no `ldcache`)	I/O speed (relative)
Default	1
`assign -F cos,sds`	7
`assign -F cos.sync,sds:3000`	9
`assign -F cos,sds.scr`	10
`assign -F sds.scr:3000`	9
`assign -F sds.scr`	3-9
(with `ldcache`)	I/O speed (relative)
Default	1
`assign -F cos,sds`	1.4
`assign -F cos.sync,sds:3000`	1.2
`assign -F cos,sds.scr`	1.2
`assign -F sds.scr:3000`	1.2
`assign -F sds.scr`	0.5-1.2

13.4.3 Using MR/SDS combinations

You can use the `sds` layer and `ldcache` in conjunction with the `mr` layer. For example, to allocate 2048 Mbytes (512 sectors) of main memory for the file, with the remainder on SSD, use the following `assign`(1) command:

```
assign -F mr.scr:512:512:0,sds.scr
```

The first 512 blocks of the file reside in main memory and the remainder of the blocks reside on SSD.

Generally, the combination of the `mr` and `sds` layers makes the maximum amount of high performance storage available to the program. The SSD is typically used in case the file size exceeds the estimated amount of main memory you can access.

The following timings from a CRAY Y-MP/8 system show the typical effects of optimization on the program used in Section 13.4.1, page 163. The program writes a file and reads it three times. Because it is not necessary to save the file afterward, you can use the `.scr` (scratch file) type. See Section 13.5.1, page 171, for more information about scratch files.

Command	I/O speed (relative)
(with no `ldcache`:)	
Default	1
`assign -F sds.scr`	4
`assign -F` `mr.scr:512:512:0,sds.scr`	4
(with `ldcache`:)	
Default	1
`assign -F cos,sds.scr`	1.2
`assign -F` `mr.scr:512:512:0,sds.scr`	1.2

13.4.4 Using a cache layer

The FFIO `cache` layer keeps recently used data in fixed size main memory or SDS buffers 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 Cray Research 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.

13.4.5 Preallocating file space

It is a good idea to preallocate space; this saves system overhead by making fewer system requests for allocation, and may reduce the number of physical I/O requests. You can allocate space by using the default value from the -A and -B options for the mkfs(8) command, or by using the assign(1) command with the -n option, as follows:

assign -n *sz*[*:st*] -q *ocblks*

The *sz* argument specifies the decimal number of 512-word blocks reserved for the data file. If this option is used on an existing file, *sz* 512-word blocks are added to the end of the file. The -q *ocblks* option specifies the number of 512-word blocks to be allocated per file system partition. These options are generally used with the -p option to do user-level striping. The *st* (*stride*) argument to the -n option is obsolete and should not be used; it specifies the allocation increment when allocating *sz* blocks.

> **Note:** For immediate preallocation, use the setf(1) command because assign does not preallocate space until the file is opened.

Use the -c option on the assign or setf command to get contiguous allocation of space so that disk heads do not have to reposition themselves as frequently. It is important to note that if contiguous allocation is unavailable, the request fails and the process might abort also.

Generally, most users should not do user-level striping (the -p option on the assign and setf commands), because it requires disk head seek operations on multiple devices. Only jobs performing I/O with large record lengths can benefit from user-level striping. Large records are those in excess of several times the size of IOS read-ahead/write-behind segments (this varies with the disk device, but it is usually at least 16 sectors), or several times the disk track size (this varies with the disk device). In addition, asynchronous I/O has a much higher payoff with user-level striping than synchronous I/O.

The assign and setf commands have a partition option, -p, that is very important for applications that perform multifile asynchronous I/O. By placing different files on different partitions (which must be on different physical devices), multiple I/O requests can be made from a job, thus increasing the I/O bandwidth to the job. The -c and -p options have no effect without the -n option.

13.4.6 User striping

When a file system is composed of partitions on more than one disk, major performance improvements can result from using the disks at the same time. This technique is called *disk striping*.

For example, if the file system spans three disks, partitions 0, 1, and 2, it may be possible to increase performance by spreading the file over all three equally. Although 300 sequential writes may be required, only 100 must go to each disk, and the disks may be writing simultaneously. You can specify striping in the following two ways, using the assign command:

```
assign -p 0-2 -n 300 -q 48 -b 144  f:filename
assign -p 0:1:2 -n 300 -q 48 -F cos:144 f:filename
```

The previous example also specifies a larger buffer size (144), which is three tracks (one per disk) if there are 48 sectors per track.

Using the bufa layer enhances the usefulness of user striping because bufa issues asynchronous I/O system calls, which are handled more efficiently by the kernel for user-striped files. In addition, the double buffering helps load balance the CPU and I/O processing. Using the previous example, better performance could be obtained from the bufa layer by using the following:

```
assign -p 0-2 -n 1000 -q 48 -F bufa:144:6
```

or

```
assign -p 0-2 -n 1000 -q 16 -F bufa:48:6
```

See Section 11.3.3, page 109, for information about the bufa layers.

Other factors, such as channel capacity, may limit the benefit of striping. Disk space on each partition should be contiguous and preallocated for maximum benefit.

Use striping only for very large records because all of the disk heads must do seeks on every transfer.

Use the df(1) command to list the partitions of a file system. For more information about the df command, see the *UNICOS User Commands Reference Manual*, publication SR–2011.

13.5 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.

13.5.1 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, such as /tmp, secondary data segments (SDS), or /ssd.

When optimizing, you should avoid writing the data to disk. This is especially important if most of the data can be held in SDS or 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. The following assign commands are examples of using these options:

```
assign  -t  f:filename
assign  -F mr.scr   f:filename
assign  -F sds.scr   f:filename
assign  -F cos,sds.scr   f:filename
```

You can think of the program's file as a scratch file and avoid flushing it at CLOSE by using the following command:

```
assign -F mr.scr u:1
```

Figure 6 shows the program's current data movement:
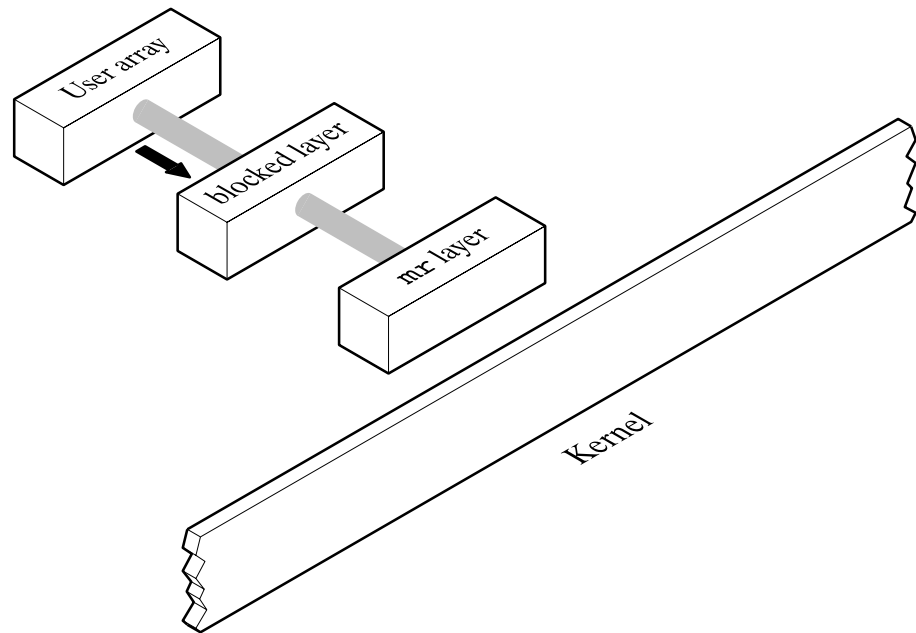
a10847

Figure 6. I/O data movement (current)

The following `procview` report shows the difference in I/O times; the last two lines of the report indicate that both the Fortran WRITE statement time and system I/O write () time were reduced to 0.

```
=====================================================================

    Fortran Unit Number      1
    File Name                fort.1
    Command Executed         a.out
    Date/Time at Open        09/04/91 17:31:38
    System File Descriptor   -1
    Type of I/O              sequential unformatted
    File Structure           COS blocked - 'blocked'
    Assign attributes        -F blocked,mr.scr
```

```
    Fortran I/O    Count of        Real
    Statement     Statements       Time
    ------------ ---------- --------------
    READ              6000        .1622
    WRITE             2000        .0862
    REWIND               3        .0005
    CLOSE                1        .0000


     0  Bytes transferred per Fortran I/O statement
    100%  Of Fortran I/O statements did not initiate a system request


===================================================================
```

If unit 1 is declared as a scratch file by using the assign command, fort.1 will no longer exist after program execution.

13.5.2 Alternate file structures

Because the original procview report indicates that no BACKSPACE was done on the file, the program might not depend on the blocked structure. Perhaps the program reads all of the data that is on every record. If it does, you can avoid using the blocked structure and save more time. Even if you cannot be sure that you do not need the blocked structure, you can still try it by using this command:

```
assign -F mr.scr u:1
```

The program will probably fail if it does require blocked structure. If it runs successfully, you will notice that it runs faster. The layer of library processing that does the record keeping was eliminated, and the program's memory use now looks like that in Figure 7.

a10848
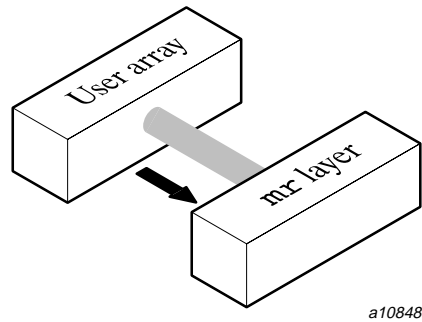
Figure 7. I/O processing with library processing eliminated

The program is now much faster. The time saved by using the `assign` commands described in this section is as follows:

Command	Speed
Default	4.6 Mbyte/s
`assign -F` `blocked,mr::1961`	27.7 Mbyte/s × 6 speedup
`assign -F` `blocked,mr.scr`	129.3 Mbyte/s × 28 speedup

Total optimization impact is I/O that is 15 times faster.

You may not see these exact improvements because many variables (such as configurations) exist that affect timings.

13.5.3 Using the asynchronous COS blocking layer

When writing a sequential COS blocked file, the library usually waits until its buffer is full before initiating a system request to write the data to the physical device. When the system request completes, the library resumes processing the user request.

The FFIO asynchronous COS layer divides this buffer in half and begins a write operation when the first half is full, but it continues processing the user request in the second half of the buffer while the system is writing data from the first half. When reading, the library tries to read ahead into the second half of the buffer to reduce the time the job must wait while waiting for system requests. This can be twice as fast as sequential I/O requests.

The asynchronous COS layer is specified with the `assign -F` command, as follows:

```
assign  -F cos.async  f:filename
assign  -F cos.async:96  f:filename
```

The second `assign` command specifies a larger buffer because the library requests (half the specified buffer size) should be the disk track size, which is assumed to be 48 sectors.

13.5.4 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.

To utilize asynchronous read-ahead and write-behind with ER90 tape files, you can use the `bufa` and the `er90` layers, as in the following example:

```
assign -F bufa,er90 f:filename
```

The `bufa` layer must be used with the `er90` layer because it supports file types that are not seekable. The `bufa` layer can also be used with disk files, as in the following example:

```
assign -F bufa:48:10 f:filename
```

This command specifies the same buffer configuration as the previous `cachea` example. The `bufa` layer uses all its pages for asynchronous read-ahead and write-behind. When writing, each page is asynchronously flushed as soon as it is full.

13.5.5 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
```

13.6 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 12, page 125.

13.7 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.

13.7.1 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. The df -p command lists thresholds for big file allocations. 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 UNICOS level

- Type of file (external or internal)

- Type of access (sequential or direct)

- Type of formatted I/O (edit-directed, list-directed, or namelist)

On UNICOS systems, the RECL parameter on the OPEN statement is accepted by the Fortran library for sequential access files. For a sequential access file, RECL is defined as the maximum record size that can be read or written. Thus, the RECL parameter on the OPEN statement can be used to adjust the maximum length of formatted records that can be read or written for that file.

If RECL is not specified, the following default maximum record lengths apply:

	Input	Output
Edit-directed formatted I/O	267	267
List-directed formatted I/O	267	133
Namelist I/O	267	133
Internal I/O	none	none
ENCODE/DECODE	none	none

13.7.2 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 sector boundary to prevent I/O performance degradation.

13.8 Other optimization options

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

13.8.1 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, page 41, for details about pipes.

13.8.2 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)

- Asynchronous queued I/O statements (AQIO)

- FFIO `cos` blocking asynchronous layer (available on IRIX systems)

- FFIO `cachea` layer (available on IRIX systems)

- FFIO `bufa` layer (available on IRIX systems)

13.9 Optimization on UNICOS/mk systems

The information in this section describes some optimization guidelines for
UNICOS/mk systems. For more information about optimization on
UNICOS/mk systems, see the *CRAY T3E Fortran Optimization Guide*, publication
SG–2518.

- Choose the largest possible transfer sizes: Using large transfer sizes
  alleviates the longer system call processing time.

- Check the MAXASYN settings: An application can become limited by the MAXASYN settings on the host machine. The default value of 35 asynchronous I/O structures limits you to 17 outstanding asynchronous I/O requests. The system administrator can view the current settings by using the crash command. The values to be checked are in the var structure; the fields that may need to be changed are v_pbuf, v_asyn, and v_maxasyn. These values can be changed by changing the values for NPBUF, NASYN, and MASAXYN in config.h.

- Coordinate PEs performing I/O: When creating files by using a UNICOS/mk application and if raw (unbuffered) I/O performance is expected, you must coordinate the PEs doing the I/O so the write requests are issued sequentially. If the PEs issue the I/O at their own speed, the host will interpret this as a non-sequential extension of a file. When this occurs, the host uses the system buffer cache to zero the space between the old EOF and the new I/O request.

- Resequence I/O when converting applications: When converting sequential applications to run on the UNICOS/mk system, resequence the I/O (from a disk perspective) by user striping the file across *N* tracks with *N* PEs performing all of the I/O, where a single PE will stride through the file by *N* records. The following diagram shows how the record numbers are assigned to the disk slices of a filesystem and shows how the PE will be performing the I/O request:

Slice A/PE-X	Slice B/PE-Y	~	Slice C/PE-Z
1	2		N
N+1	N+2		2N
2N+1	2N+2		3N
~	~	~	~
K*N+1	K*N+2		(K+1)*N

- Use CF90 and IEEE data conversion facilities: When an unformatted Cray PVP data file is to be read on the Cray MPP system, write a conversion program to run on the Cray PVP system that uses the CF90 compiler and the T3D data conversion layer. For data files that have integer elements, no conversion is necessary. For data files that have real or logical elements, use an assign -N t3d statement for the output data file.

FFIO Layer Reference   [14]

This chapter provides details about each of the following FFIO layers. **An asterisk (*) indicates that the layer is available on IRIX systems**:

Layer	Definition
`blankx` or `blx`	Blank compression/expansion layer
`bmx` or `tape`	UNICOS online tape handling
`bufa` *	Library-managed asynchronous buffering
`c205`	CDC CYBER 205 record formats
`cache`*	`cache` layer
`cachea` *	`cachea` layer
`cdc`	CDC 60-bit NOS/SCOPE file formats
`cos` *	COS blocking
`er90`	ER90 handling
`event`	I/O monitoring (not available on CRAY T3E systems)
`f77` *	UNIX record blocking
`fd`*	File descriptor
`global`	Cache distribution layer (CRAY T3E systems only)
`ibm`	IBM file formats
`mr`	Memory-resident file handlers
`nosve`	CDC NOS/VE file formats
`null` *	The `null` layer
`sds`	SDS resident file handlers (not available on CRAY T3E systems)
`syscall` *	System call I/O
`system` *	Generic system layer
`text` *	Newline separated record formats
`user`* and `site` *	Writable layer

vms   VAX/VMS file formats

14.1 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. This functionality is available if your application is linked with Cray
Research's CrayLibs 3.0 or later release. See the INTRO_FFIO(3F) man page for
more details about FFIO layers.

14.2 Individual layers

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

14.2.1 The `blankx` expansion/compression layer (not available on IRIX systems)

The `blankx` or `blx` layer performs blank compression and expansion on a stream of 8-bit characters. The syntax for this layer is as follows:

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

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

The keyword specification for this layer is as follows:

```
blankx.[type][.blxchr=num1][.blnk=num2]
```

```
blx.[type][.blxchr=num1][.blnk=num2]
```

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

Value	Definition
cos	COS-style blank compression . (`blxchr`= 27 or 0x1D)
ctss	CTSS-style blank compression. (`blxchr`= 48 or 0x30)
c205	CYBER 205–style blank compression. (`blxchr`= 48 or 0x30)

The *num1* field contains the decimal value of the ASCII character used as the escape code to control the blank compression.

The *num2* field contains the decimal value of the ASCII character that is the object of the compression. This is usually the ASCII blank (0x20).

Table 10. Data manipulation: `blankx` layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Filter. Takes characteristics of lower-level layer but does some data transformation.	No	`blx` specific

Table 11. Supported operations: `blankx` layer

Operation	Supported operations		Required of next lower level?	
	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	Only in lower-level layer	Yes	Only if explicitly requested
`ffweod`	Passed through	Only in lower-level layer	Yes	Only if explicitly requested
`ffseek`	No	Only seek `(fd,0,0)` for rewind	Yes	Only on rewind
`ffpos`	Yes		NA	NA
`ffbksp`	Passed through	Only in lower-level layer	Yes	Only if explicitly requested

14.2.2 The `bmx/tape` layer (deferred implementation on IRIX systems)

The `bmx` or `tape` layer handles the interface to online magnetic tapes. The `bmx` layer uses the tape list I/O interface on Cray Research systems.

A magnetic tape does not use control words to delimit records and files; however, control words are part of the physical representation of the data on the medium. On a magnetic tape, each tape block is considered a record.

The following is the syntax for this layer:

```
bmx:[num1]:[num2]
```

```
tape:[num1]:[num2]
```

The keyword specification is as follows:

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

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

The *num1* argument specifies the size in 512-word blocks for each buffer. The *num2* argument specifies the number of buffers to use.

The `bmx` layer may be used with ER90 files that have been mounted in `blocked` mode. The ER90 device places restrictions on the amount of data that can be written to a tape block; see the *UNICOS Tape Subsystem User's Guide*, publication SG–2051, for details.

Table 12 describes the EOF and EOD behavior of the `bmx` layer.

Table 12. `-T` specified on `tpmnt`

Type of tapes	EOF/EOD	No	Yes
Labeled	EOF	Never returned	At user tape marks
	EOD	At end-of-file	At label/end-of-file

Type of tapes	EOF/EOD	No	Yes
Unlabeled	EOF	Never returned	At user tape marks
	EOD	At double tape mark	Never returned

The EOF label is always considered an EOD. For unlabeled tapes without the -T option specified, nothing can be considered an EOF. The double tape mark shows the EOD. For unlabeled tapes specified with -T, nothing can be considered an EOD and every tape mark is returned as an EOF.

No optional fields are permitted.

Table 13. Data manipulation: `bmx/tape` layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Record with multiple EOF if users specify with `tpmnt -T`	Yes	`bmx` specific

Table 14. Supported operations: `bmx/tape` layer

Operation	**Supported**	**Comments**
`ffopen`	Yes	
`ffread`	Yes	
`ffreada`	Yes	Always synchronous
`ffreadc`	Yes	
`ffwrite`	Yes	
`ffwritea`	Yes	Always synchronous
`ffwritec`	Yes	
`ffclose`	Yes	

Operation	Supported	Comments
ffflush	Yes	
ffweof	Yes	Writes tape mark if allowed
ffweod	Yes	
ffseek	No	seek (fd,0,0) only (equal to rewind)
ffpos	Yes	
ffbksp	Yes	

Lower-level layers are not allowed. Exact implementation depends on operating system and hardware platform.

14.2.3 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 maximum allowed value on UNICOS and UNICOS/mk systems 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

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

Table 15. Data manipulation: `bufa` layer

Granularity	Data model	Truncate on write
1 bit (UNICOS and UNICOS/mk	Stream	No
8 bits (IRIX systems	Stream	No

Table 16. 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	
`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	

14.2.4 The CYBER 205/ETA (c205) blocking layer (not available on IRIX systems)

The c205 layer performs blocking and deblocking of the native type for the CDC CYBER 205 or ETA computer systems. The general format of the specification follows:

c205.w:[*recsize*]:[*bufsize*]

The keyword specification follows:

c205.w[.bufsize=*num2*]

The w is CYBER 205 W-type records and must be specified. The *recsize* field should not be specified because it is reserved for future use as a maximum record size. The *bufsize* refers to the working buffer size for the layer and should be specified as a nonnegative decimal number (in bytes).

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

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

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

Table 17. Data manipulation: c205 layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Record	Yes. CDC end-of-group delimiter (EOG) maps to EOF, and CDC end-of-file (EOF) maps to EOD.	x records

Table 18. Supported operations: `c205` 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-op	No	
`ffweof`	Yes	Mapped to end-of-group	No	
`ffweod`	Yes	Mapped to end-of-file	Yes	
`ffseek`	Yes	`seek(fd,0,0)` only (equals rewind)	Yes	Requires that the underlying interface be a stream
`ffpos`	Yes		NA	
`ffbksp`	No		No	

14.2.5 The `cache` layer

The `cache` layer allows recently accessed parts of a file to be cached either in main memory or in a secondary data segment (SDS). 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 can be either mem or sds (.sds is not allowed on IRIX systems or on CRAY T3E systems). mem directs that cache pages reside in main memory; sds directs that the pages reside in secondary data segments (SDS). *num1* specifies the size, in 4096–byte blocks, of each cache page buffer. The default is 8. The maximum allowed value for *num1* on IRIX systems is 32,767. The maximum allowed value on UNICOS and UNICOS/mk systems 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

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* on IRIX systems is *num3=num1*. On UNICOS and UNICOS/mk systems, the default is *num3=num1×num2*.

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 19. Data manipulation: cache layer

Granularity	Data model	Truncate on write
1 bit (UNICOS and UNICOS/mk systems)	Stream (mimics UNICOS system calls)	No
8 bit (IRIX systems)	Stream	No
512 words (cache.sds)	Stream	No

Table 20. 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
ffpos	Yes		NA	
ffbksp	No		NA	

14.2.6 The `cachea` layer

The `cachea` layer allows recently accessed parts of a file to be cached either in main memory or in a secondary data segment (SDS). 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[type]:[num1]:[num2]:[num3]:[num4]
```

The keyword syntax is as follows:

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

type   Directs that cache pages reside in main memory (mem) or SDS (sds). SDS is available only on UNICOS systems.

num1   Specifies the size, in 4096-byte blocks, of each cache page buffer. Default is 8. The maximum allowed value for *num1* on IRIX systems is 32,767. The maximum allowed value on UNICOS and UNICOS/mk systems 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

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.

On IRIX systems, stacked shared cachea layers are not supported. On UNICOS and UNICOS/mk systems, stacked shared layers are supported, but in multitasked programs, different files must not mix the order of the shared caches.

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
```

- The following specifications can both be used by a multitasked program on UNICOS systems:

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

Table 21. Data manipulation: `cachea` layer

Granularity	Data model	Truncate on write
1 bit (UNICOS and UNICOS/mk systems	Stream (mimics UNICOS system calls)	No
8 bit (IRIX systems	Stream (mimics UNICOS system calls)	No

Table 22. 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	
`ffweof`	No		No	
`ffweod`	Yes		Yes	

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

14.2.7 The `cdc` layer (not available on IRIX systems)

The `cdc` layer handles record blocking for four common record types from the 60-bit CYBER 6000 and 7000 series computer systems, which run the CDC 60-bit NOS, NOS/VE, or SCOPE operating system. The general format of the specification follows:

cdc[`.recfmt`]`.`[`tpfmt`]

There is no alternate keyword specification for this layer.

The supported *recfmt* values are as follows:

Values	Definition
iw	I-type blocks, W-type records
cw	C-type blocks, W-type records
cs	C-type blocks, S-type records
cz	C-type blocks, Z-type records

The *tpfmt* field can have one of the following three values that indicate the presence of block trailers and other low-level characteristics.

Field	Definition
disk	Disk type structure, for use with station transfers of CYBER data
i	NOS internal tape format

si   System internal or SCOPE internal tape format

Note: The i and si fields require a lower-level layer that handles records. A stream model in the lower-level layers does not work.

The disk field requires a lower layer that handles records when endfile makes exist prior to the end of data.

Table 23. Data manipulation: cdc layer

Granularity	Data model	Truncate on write	Implementation strategy
6 bits for cz records, 1 bit for iw records, and 60 bits for cs and cw records.	Record	Yes	cdc specific

Table 24. Supported operations: cdc 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-op	No	
ffweof	Yes		No	

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

14.2.8 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.

When writing, full buffers are written in full record mode, specifically so that the magnetic tape bmx layer can be used on the system side to read and write COS transparent tapes. To choose the block size of the tape, select the buffer size.

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 25. Data manipulation: `cos` layer

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

Table 26. 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	

Note: `seek` operations are supported only to allow for rewind (`seek(fd,0,0)`), seek-to-end (`seek(fd,0,2)`) and a GETPOS(3F) or SETPOS(3F) operation, where the position must be on a record boundary.

GETPOS(3F) and SETPOS(3F) are not available on IRIX systems.

14.2.9 The `er90` layer (available only on UNICOS systems)

The `er90` layer is not supported on IRIX systems or on CRAY T3E systems. It is available only on UNICOS systems.

The `er90` layer handles the interface to the ER90 files. No arguments are accepted.

Table 27. Data manipulation: `er90` layer

Granularity	Data model	Truncate on write
8 bits	Stream	Yes

Table 28. Supported operations: `er90` layer

Operation	Supported	Comments
ffopen	Yes	
ffread	Yes	
ffreada	Yes	
ffreadc	No	
ffwrite	Yes	
ffwritea	Yes	
ffwritec	Yes	
ffclose	Yes	
ffflush	Yes	
ffweof	No	
ffweod	Yes	

Operation	Supported	Comments
ffseek	Yes	`ffseek(fd,0,0)` only (equals rewind)
ffbksp	No	

Lower-level layers are not allowed.

14.2.10 The `event` layer (deferred implementation on IRIX systems)

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
```

On CRAY T3E systems, if more than one PE is using the `event` layer, and you set the `FF_IO_LOGFILE` environment variable, you must use the plus sign (+) before the file name to prevent PE *a* from overwriting the information written by PE *b*. Using the plus sign also means that the information will be appended to an existing file.

On CRAY T3E systems, you can also use the `FF_IO_LOGFILEPE` environment variable to name a file to which statistics are written. The file name will be *x.n*, where *x* is the name specified by the environment variable and *n* is the number of the PE which wrote the file. The default action is to overwrite the existing file. To append information to an existing file, specify a plus sign (+) before the file name.

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.

14.2.11 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 29. Data manipulation: `f77` layer

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

Table 30. 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	

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
ffflush	Yes		No	
ffweof	Passed through		Yes	Only if explicitly requested
ffweod	Yes		Yes	
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	

14.2.12 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 Section 14.2.20, page 223, for details.

14.2.13 The `global` layer (deferred implementation on IRIX systems)

The `global` layer is a caching layer that distributes data across all processing elements (PEs) on CRAY T3E systems. Open and close operations require

participation by all PEs; all other operations are independently performed by one or more PEs.

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 PE or `globpos` (deferred implementation), in which the file position is global to all PEs.

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 PE. If there are *n* PEs, then $n \times num2$ cache pages are used.

Table 31. Data manipulation: `global` layer

Granularity	Data model	Truncate on write
8 bits	Stream	No

Table 32. Supported operations: `global` 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	

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
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	

14.2.14 The `ibm` layer (deferred implementation on IRIX systems)

The `ibm` layer handles record blocking for seven common record types on IBM operating systems. The general format of the specification follows:

ibm.[*type*]:[*num1*]:[*num2*]

The keyword specification follows:

ibm[.*type*][.recsize=*num1*][.mbs=*num2*]

The supported *type* values are as follows:

Value	Definition
u	IBM undefined record type
f	IBM fixed-length records
fb	IBM fixed-length blocked records

v	IBM variable-length records
vb	IBM variable-length blocked records
vbs	IBM variable-length blocked spanned records

The f format is fixed-length record format. For fixed-length records, *num1* is the fixed record length (in bytes) for each logical record. Exactly one record is placed in each block.

The fb format records are the same as f format records except that you can place more than one record in each block. *num1* is the length of each logical record. *num2* must be an exact multiple of *num1*.

The v format records are variable-length records. *recsize* is the maximum number of bytes in a logical record. *num2* must exceed *num1* by at least 8 bytes. Exactly one logical record is placed in each block.

The vb format records are variable-length blocked records. This means that you can place more than one logical record in a block. *num1* and *num2* are the same as with v format.

The vbs format records have no limit on record size. Records are broken into segments, which are placed into one or more blocks. *num1* should not be specified. When reading, *num2* must be at least large enough to accommodate the largest physical block expected to be encountered.

The *num1* field is the maximum record size that may be read or written. The vbs record type ignores it.

The *num2* (maximum block size) field is the maximum block size that the layer uses on reads or writes.

Table 33. Values for maximum record size on ibm layer

Field	Minimum	Maximum	Default	Comments
u	1	32,760	32,760	
f	1	32,760	None	Required
fb	1	32,760	None	Required
v	5	32,756	32,752	Default is *num2*-8 if not specified

Field	Minimum	Maximum	Default	Comments
vb	5	32,756	32,752	Default is *num2*-8 if not specified
vbs	1	None	None	No maximum record size

Table 34. Values for maximum block size in `ibm` layer

Field	Minimum	Maximum	Default	Comments
u	1	32,760	32,760	Should be equal to *num1*
f	1	32,760	*num1*	Must be equal to *num1*
fb	1	32,760	*num1*	Must be multiple of *num1*
v	9	32,760	32,760	Must be >= *num1* + 8
vb	9	32,760	32,760	Must be >= *num1* + 8
vbs	9	32,760	32,760	

Table 35. Data manipulation: `ibm` layer

Granularity	Data model	Truncate on write	Implementation strategy
8 bits	Record	No for `f` and `fb` records. Yes for `v`, `vb`, and `vbs` records.	`f` records for `f` and `fb`. `v` records for `u`, `v`, `vb`, and `vbs`.

Table 36. Supported operations: `ibm` layer

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
ffopen	Yes		Yes	
ffread	Yes		Yes	

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

14.2.15 The mr layer (deferred implementation on IRIX systems)

The memory-resident (mr) layer lets users declare that a file should reside in memory. The mr layer tries to allocate a buffer large enough to hold the entire file.

The options are as follows:

mr[.*type*[.*subtype*]]:*num1*:*num2*:*num3*

The keyword specification is as follows:

```
mr[.type[.subtype]][.start_size=num1][.max_size=num2]
    [.inc_size=num3]
```

The *type* field specifies whether the file in memory is intended to be saved or is considered a scratch file. This argument accepts the following values:

Value	Definition
save	Loads (reads) as much of the file as possible into memory when the file is opened (if it exists). If the data in memory is changed, the file data is written back to the next lower layer at close time. The save option also modifies the behavior of overflow processing. save is the default.
scr	Does not try to load at open and discards data on close (scratch file). The scr option also modifies the behavior of overflow processing.

The *subtype* field specifies the action to take when the data can no longer fit in the allowable memory space. It accepts the following values:

Value	Definition
ovfl	Excess data that does not fit in the specified medium is written to the next lower layer. ovfl is the default value.
novfl	When the memory limit is reached, any further operations that try to increase the size of the file fail.

The *num1*, *num2*, and *num3* fields are nonnegative integer values that state the number of 4096–byte blocks to use in the following circumstances:

Field	Definition
num1	When the file is opened, this number of blocks is allocated for the file. Default: 0.
num2	This is the limit on the total size of the memory space allowed for the file in this layer. Attempted growth beyond this limit causes either overflow or operation failure, depending on the overflow option specified. Default: $2^{46}-1$
num3	This is the minimum number of blocks that are allocated whenever more memory space is required to accommodate file

growth. Default: 256 for SDS files and 32 for memory resident files.

The *num1* and *num3* fields represent best-effort values. They are intended for tuning purposes and usually do not cause failure if they are not satisfied precisely as specified (for example, if the available memory space is only 100 blocks and the chosen *num3* value is 200 blocks, growth is allowed to use the 100 available blocks rather than failing to grow, because the full 200 blocks requested for the increment are unavailable).

When using the `mr` layer, large memory-resident files may reduce I/O performance for sites that provide memory scheduling that favors small processes over large processes. Check with your system administrator if I/O performance is diminished.

⚠️ **Caution:** Use of the default value for the `max` parameter can cause program failure if the file grows and exhausts the entire amount of memory available to the process. If the file size might become quite large, always provide a limit.

Memory allocation is done by using the `malloc`(3C) and `realloc`(3C) library routines. The file space in memory is always allocated contiguously.

When allocating new chunks of memory space, the *num3* argument is used in conjunction with `realloc` as a minimum first try for reallocation.

Table 37. Data manipulation: `mr` layer

Primary function	Granularity	Data model	Truncate on write
Avoid I/O to the extent possible, by holding the file in memory.	1 bit	Stream (mimics UNICOS system calls)	No

Table 38. Supported operations: `mr` layer

Operation	Supported operations		Required of next lower level?	
	Supported	Comments	Used	Comments
ffopen	Yes		Yes	Sometimes delayed until overflow
ffread	Yes		Yes	Only on open
ffreada	Yes		No	
ffreadc	Yes		No	
ffwrite	Yes		Yes	Only on close, overflow
ffwritea	Yes		No	
ffwritec	Yes		No	
ffclose	Yes		Yes	
ffflush	Yes	No-op	No	
ffweof	No	No representation	No	No representation
ffweod	Yes		Yes	
ffseek	Yes	Full support (absolute, relative, and from end)	Yes	Used in open and close processing
ffpos	Yes		NA	
ffbksp	No	No records	No	

14.2.16 The `nosve` layer (not available on IRIX systems)

The `nosve` layer handles record blocking for five common record types on CDC NOS/VE operating systems.

The general format of the specifications is as follows:

nosve[.*type*]:[*num1*]:[*num2*]

The format of the keyword specifications is as follows:

```
nosve[.type][.recsize=num1][.mbs=num2]
```

The supported *type* fields follow:

Field	Definition
v	NOS/VE format record
f	ANSI F fixed-length records
s	ANSI S format (segmented) records
d	ANSI D format (variable-length) records
u	NOS/VE undefined record

The *num1* field is the maximum record size that can be read or written. The s and v record types ignore it.

Table 39. Values for maximum record size

recfmt	Minimum	Maximum	Default	Comments
v	1	No maximum	None	
f	1	65,536	None	Required
s	1	No maximum	None	
d	1	9,995	4,123	
u	1	32,760	32,760	

Table 40. Values for maximum block size

recfmt	Minimum	Maximum	Default	Comments
v	32	Memory size	32,768	Working buffer size
f	1	65,536	*num1*	
s	6	32,767	4,128	
d	5	32,767	4,128	
u	1	32,760	32,760	

For the `nosve.v` format, the working buffer size can affect performance. If the buffer size is at least as large as the largest record that will be written, the system overhead is minimized. For the `nosve.u` record format, *num1* and *num2* are the same thing. For `nosve.f` and `nosve.d` records, the maximum block size must be at least as large as the maximum record size. You can place more than one record in a block.

For `nosve.s` records, one or more segments are placed in each block (a record is composed of one or more segments).

Table 41. Data manipulation: `nosve` layer

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

Table 42. Supported operations: `nosve` layer

| Operation | Supported operations | | Required of next lower level? | |
	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 for `s` records; passed through for others	Yes	Only if explicitly requested
ffweod	Yes		Yes	
ffseek	Yes	`ffseek(fd,0,0)` only (equals rewind)	Yes	Extensively for `nosve.v`
ffpos	Yes		NA	
ffbksp	No		No	

14.2.17 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
```

14.2.18 The `sds` layer (available only on UNICOS systems)

The `sds` layer is not available on CRAY T3E systems or on IRIX systems.

The `sds` layer lets users declare that a file should reside on SDS. The specification for this layer follows:

sds[.*type*:[*subtype*]]:[*num1*]:[*num2*]:[*num3*]

The keyword specification is as follows:

sds[.*type*[.*subtype*]][.start_size=*num1*][.max_size=*num2*]
  [.inc_size=*num3*]

The *type* field specifies whether the file to reside in SDS is intended to be saved. This field can have the following values:

Value	Definition
save	Loads (reads) as much of the file as possible into SDS as soon as the file is opened (if it exists). If the data in SDS is changed, the SDS data is written back to the next lower layer at close time.

The save option also modifies the behavior of overflow. save is the default.

scr   Does not attempt to load at open and discards data on close (scratch file). The scr option also modifies the behavior of overflow processing.

The *subtype* field specifies the action to take when the data can no longer fit in the allowable SDS space. It can have the following values:

Value	Definition
ovfl	Excess data that does not fit in the specified medium is written to the next lower layer. This is the default.
novfl	When the SDS limit is reached, any further operations that try to increase the size of the file fails.

The *num1*, *num2*, and *num3* fields are nonnegative integer values that state the number of 4096-byte blocks to use in the following circumstances.

Field	Definition
num1	When the file is opened, this number of blocks is allocated for the file.
num2	This is the limit on the total size of the SDS space allowed for the file in this layer. Attempted growth beyond this limit causes either overflow or operation failure, depending on the overflow option specified.
num3	This is the minimum number of blocks that are allocated each time more SDS space is required to accommodate file growth.

The *num1* and *num3* fields are used for tuning purposes and usually do not fail if they are not used precisely as specified. For example, if the available SDS space is only 100 blocks, and the chosen increase (*num3*) value is 200 blocks, growth is allowed to use the 100 available blocks instead of failing to grow (because the full 200 blocks requested for the increment are unavailable). Similarly, the *num3* value of 200 implies allocation in minimum size chunks of 200 blocks. If 200 blocks of contiguous space is unavailable, the allocation is satisfied with whatever space is available.

The specification for sds is equivalent to the following specification:

```
sds.save.ovfl:0:35184372088832:256
```

Overflow is provided when the requested data cannot completely reside in SDS. This can occur either because the SDS space requested from the system is not available or because the *num2* (maximum size) argument was specified.

When overflow occurs, a message prints to standard error stating the file name and the overflow size. The overflow I/O to the next lower layer depends on the *type* argument. If `save` is specified, the `sds` layer assumes that the part of the file that resides in SDS must eventually be written to the lower-level layer (usually disk).

The overflowed data is written to the lower-level layer at a position in the file that corresponds to the position it will occupy after the SDS-resident data is flushed. Space is reserved at overflow time in the file to accommodate the SDS resident part of the file.

If the `scr` option is selected, the SDS resident part of the file is considered disposable. Space for it is not reserved in the file on the lower-level layer. The overflow operations behave as though the first overflowed bit in the file is bit 0 of the lower-level layer, as in the following example:

```
# requests a max of 1 512-word block of SDS
assign -F sds.save.ovfl:0:1 fort.1
```

Assume that the file does not initially exist. The initial SDS size is 0 blocks, and the size is allowed to grow to a maximum of 1 block. If a single write of 513 words was done to this file, the first 512 words are written to SDS. The remaining word is written to file `fort.1` at word position 512.

Words 0 through 511 are not written until the `sds` layer is closed and the SDS data is flushed to the lower-level layer. Immediately after the write completes, SDS contains 512 words, and `fort.1` consists of 513 words. Only the last word contains valid data until the file is closed.

If the `assign` command is of the following form, it is assumed that the entire file is disposable if 513 words are written to the file:

```
# requests a max of 1 512-word block of SDS
assign -F sds.scr.ovfl:0:1 fort.1
```

It is not necessary to reserve space in `fort.1` for the SDS data. When the 513 words are written to the file, the first 512 words are written to SDS. The 513th word is written to word 0 of `fort.1`. After the completion of the write, `fort.1` consists of 1 word. The `fort.1` file is deleted when the file is closed.

SDS allocation is done through the `sdsalloc`(3F) library routine. The file space in SDS is allocated (as far as possible) in a contiguous manner, but if contiguous

space is not found, any available fragments are used before overflow is forced on any file.

When allocating **new** chunks of SDS space, the *num3* argument is used as a minimum first try for allocation.

Table 43. Data manipulation: `sds` layer

Primary function	Granularity	Data model	Truncate on write
The `sds` layer lets the users obtain the fastest possible I/O rates through the SDS *hot path*.	1 bit	Stream (mimics UNICOS system calls)	No

Table 44. Supported operations: `sds` layer

	Supported operations		Required of next lower level?	
Operation	**Supported**	**Comments**	**Used**	**Comments**
`ffopen`	Yes		Yes	Sometimes delayed until overflow
`ffread`	Yes		Yes	Only on open
`ffreada`	Yes		No	
`ffreadc`	Yes		No	
`ffwrite`	Yes		Yes	Only on close, overflow
`ffwritea`	Yes		No	
`ffwritec`	Yes		No	
`ffclose`	Yes		Yes	
`ffflush`	Yes	Flushes only internal buffer not SDS	No	
`ffweof`	No		No	No representation
`ffweod`	Yes	No representation	Yes	

	Supported operations		Required of next lower level?	
Operation	**Supported**	**Comments**	**Used**	**Comments**
ffseek	Yes	Full support (absolute, relative, and from end)	Yes	Used in open and close processing
ffpos	Yes		NA	
ffbksp	No	No records	No	No records

14.2.19 The `syscall` layer

The `syscall` layer directly maps each request to an appropriate system call. The layer does not accept any options.

Table 45. Data manipulation: `syscall` layer

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

Table 46. Supported operations: `syscall` layer

Operation	**Supported**	**Comments**
ffopen	Yes	open
ffread	Yes	read
ffreada	Yes	reada
ffreadc	Yes	read plus code
ffwrite	Yes	write
ffwritea	Yes	writea
ffwritec	Yes	write plus code
ffclose	Yes	close

Operation	Supported	Comments
ffflush	Yes	None
ffweof	No	None
ffweod	Yes	trunc(2)
ffseek	Yes	lseek(2)
ffpos	Yes	
ffbksp	No	

Lower-level layers are not allowed.

14.2.20 The system layer

The system layer is implicitly appended to all specification lists, if not explicitly added by the user (unless the syscall, tape, er90, 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.

The layer does not accept any options. Lower-level layers are not allowed.

14.2.21 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 47. Data manipulation: text layer

Granularity	Data model	Truncate on write
8 bits	Record.	No

Table 48. 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	
ffreadc	Yes		No	
ffwrite	Yes		Yes	
ffwritea	Yes	Always synchronous	No	
ffwritec	Yes		No	
ffclose	Yes		Yes	

	Supported operations		Required of next lower level?	
Operation	Supported	Comments	Used	Comments
ffflush	Yes		No	
ffweof	Passed through		Yes	Only if explicitly requested
ffweod	Yes		Yes	
ffseek	Yes		Yes	
ffpos	Yes		No	
ffbksp	No		No	

14.2.22 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 15, page 231 for an example of how to create an FFIO layer.

14.2.23 The `vms` layer (deferred implementation on IRIX systems)

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 49. 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 50. 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 51. 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 52. Supported operations: `vms` layer

Operation	Supported operations		Required of next lower level?	
	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	
`ffseek`	Yes	`seek(fd,0,0)` only (equals rewind)	Yes	`seek(fd,0,0)` only

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

Creating a `user` Layer   [15]

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. Section 15.2, page 234, is an example of a `user` layer.

15.1 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.

15.1.1 The operations structure

Many of the operations try to mimic the UNICOS 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:

Available on UNICOS, UNICOS/mk and IRIX systems:

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

Available on UNICOS and UNICOS/mk systems only:

```
nb = ffreada(fd, buf, nb, stat, fulp, &ubc);
nb = ffreadc(fd, buf, nb, stat, fulp);
nb = ffwrite(fd, buf, nb, stat, fulp, &ubc);
nb = ffwritea(fd, buf, nb, stat, fulp, &ubc);
nb = ffwritec(fd, buf, nb, stat, fulp);
ret = ffclose(fd, stat);
ret = ffflush(fd, stat);
ret = ffweof(fd, stat);
ret = ffweod(fd, stat);
ret = ffbksp(fd, stat);
```

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.

15.1.2 FFIO and the stat structure

The *stat* structure contains four fields in the current implementation. They
mimic the *iosw* structure of the UNICOS 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.

15.2 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 Section 15.1.1, page 232, and Section 15.1.2, page 233. 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. The following script fragment is suggested for UNICOS systems:

```
# -D_LIB_INTERNAL is required to obtain the
#  declaration of struct fdinfo in <ffio.h>
#
cc -c -D_LIB_INTERNAL -hcalchars usr*.c
cat usr*.o > user.o
#
# Note that the -F option is selected that loads
# and links the entries despite not having any
# hard references.

segldr -o abs -F user.o myprog.o
assign -F user,others... fort.1
./abs
```

For CRAY T3E systems, replace the segldr command with the following:

```
f90 main.f user.o -Wl"-D select(user)=yes"
```

On IRIX systems, to build for 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 CRAY RESEARCH, INC.
*    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 CRAY RESEARCH, INC.
 *  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 CRAY RESEARCH, INC.
 *  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 CRAY RESEARCH, INC.
 *  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 CRAY RESEARCH, INC.
 *  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 CRAY RESEARCH, INC.
 *  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 CRAY RESEARCH, INC.
 *  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 UNICOS 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 CRAY RESEARCH, INC.
 *  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
* UNICOS 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 UNICOS
 *  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 CRAY RESEARCH, INC.
 *  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);
```

Older Data Conversion Routines   [A]

The UNICOS library contains newer conversion routines for the following foreign types:

Type	Routines
IBM	IBM2CRAY(3F), CRAY2IBM
CDC	CDC2CRAY(3F), CRAY2CDC
VAX/VMS	VAX2CRAY(3F), CRAY2VAX
NOS/VE	NVE2CRAY(3F), CRAY2NVE
ETA/CYBER 205	ETA2CRAY(3F), CRAY2ETA
IEEE	IEG2CRAY(3F), CRAY2IEG

The charts in this appendix list the older foreign data conversion routines that Cray Research supports for compatibility. The following abbreviations are used: int. (integer), f.p. (floating-point number), s.p. (single-precision number), and d.p. (double-precision number). Brackets in the synopsis indicate an optional parameter; it may be omitted. See the *Application Programmer's Library Reference Manual*, publication SR–2165, for a complete description of each routine.

A.1 Old IBM data conversion routines

The following lists IBM data conversion for integer, single-precision, double-precision, logical, and character data:

Convert IBM to/from CRI	Synopsis
INTEGER*1	CALL USICTC (*src*, *isb*, *dest*, *num*, *len* [, *inc*])
INTEGER*4 / 64-bit int.	CALL USICTI (*src*, *dest*, *isb*, *num*, *len* [, *inc*])
Pack decimal / 64-bit int.	CALL USICTP (*ian*, *dest*, *isb*, *num*) CALL USPCTC (*src*, *isb*, *num*, *ian*)
32-bit f.p. / 64-bit s.p.	CALL USSCTC (*dpn*, *isb*, *dest*, *num* [, *inc*]) CALL USSCTI (*fpn*, *dest*, *isb*, *num*, *ier* [, *inc*])

Convert IBM to/from CRI	Synopsis
64-bit d.p. / 64-bit s.p.	CALL USDCTC (*dpn, isb, dest, num* [, *inc*]) CALL USDCTI (*fpn, dest, isb, num, ier* [, *inc*])
LOGICAL*1	CALL USLCTC (*src, isb, dest, num, len* [, *inc*])
LOGICAL*4 / 64-bit log.	CALL USLCTI (*src, dest, isb, num, len* [, *inc*])
EBCDIC /ASCII	CALL USCCTC (*src, isb, dest, num, npw* [, *val*]) CALL USCCTI (*src, dest, isb, num, npw* [, *val*])

For UNICOS and UNICOS/mk IEEE systems, CRI2IBM(3F) and IBM2CRI(3F)
provide all of the functionality of the preceding routines.

A.2 Old CDC data conversion routines

The following lists CDC data conversion routines for single-precision numbers
and character data:

Convert CDC to/from CRI	Synopsis
60-bit s.p. /64-bit s.p.	CALL FP6064 (*fpn, dest, num*) CALL FP6460 (*fpn, dest, num*)
Display Code / ASCII	CALL DSASC (*src, sc, dest, num*) CALL ASCDC (*src, sc, dest, num*)

A.3 Old VAX/VMS data conversion routine

The following lists VAX/VMS data conversion routines for integer,
single-precision, double-precision, complex, and logical data:

Convert VAX/VMS to/from CRI	Synopsis
INTEGER*2	CALL VXICTC (*in, isb, dest, num, len* [, *inc*])
INTEGER*4 / 64-bit int.	CALL VXICTI (*in, dest, isb, num, len* [, *inc*])
32-bit F format / 64-bit s.p.	CALL VXSCTC (*fpn, isb, dest, num* [, *inc*]) CALL VXSCTI (*fpn, dest, isb, num, ier* [, *inc*])

64-bit D format ∕ 64-bit s.p.	CALL VXDCTI (*fpn, dest, isb, num, ier* [, *inc*]) CALL VXDCTC (*dpn, isb, dest, num* [*,inc*])
64-bit G format ∕ 64-bit s.p.	CALL VXGCTC(*dpn, isb, dest, num* [, *inc*]) CALL VXGCTI (*fpn, dest, isb, num, ier* [, *inc*])
64-bit complex ∕complex	CALL VXZCTC (*dpn, isb, dest, num* [, *inc*]) CALL VXZCTI (*fpn, dest, isb, num, ier* [, *inc*])
Logical ∕ 64-bit logical	CALL VXLCTC (*src, isb, dest, num, len* [, *inc*])

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).

The `/etc/rc`(8) script is the shell procedure that mounts file systems and activates accounting, error logging, and system activity logging. It is a major script that is called by the `init`(8) command in bringing UNICOS from single-user to multiuser mode. The `/etc/rc.local` script is provided on UNIX systems to allow site modification of the start-up sequence.

A tree-structured collection of files and their associated data and attributes. A file system is mounted to connect it to the overall file system hierarchy and make it accessible.

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.

sector

A part of the format scheme in disk drives. A disk drive is composed of equal segments called sector; a sector is the smallest unit of transfer to or from a disk drive. The size of a sector depends on the disk drive. See also `block`.

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. (2) On CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP systems, a contiguous storage address space on a physical device, specified by a starting cylinder and number of blocks.

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