Fortran Application Programmer’s
I/O Reference Manual
S–2351–50
New Features

This manual supports the Programming Environment 5.0 releases for the Cray X1 system. The following changes have been made from the last release of this document:

- Content that is covered in greater detail in the Fortran reference manuals has been removed to eliminate redundancy.
- This manual has been reorganized to improve the focus on the assign environment and flexible file I/O (FFIO) functions.
<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
</table>
| 4.1     | August 12, 2002  
Draft version to support the Cray Programming Environment 4.1 release. |
| 4.2     | December 20, 2002  
Draft version to support the Cray Programming Environment 4.2 release. |
| 4.3     | March 31, 2003  
Draft version to support the Cray Programming Environment 4.3 release. |
| 5.0     | June 2003  
Supports the Cray Programming Environment 5.0 releases. |
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>ix</td>
</tr>
<tr>
<td>Accessing Cray Documentation</td>
<td>ix</td>
</tr>
<tr>
<td>Error Message Explanations</td>
<td>x</td>
</tr>
<tr>
<td>Typographical Conventions</td>
<td>xi</td>
</tr>
<tr>
<td>Ordering Documentation</td>
<td>xi</td>
</tr>
<tr>
<td>Reader Comments</td>
<td>xii</td>
</tr>
</tbody>
</table>

### Introduction [1]

#### Using the Assign Environment [2]

- **assign Basics** .................................................. 3
  - Assign Objects and Open Processing .......................... 4
  - The `assign` Command .......................................... 5
  - Assign Library Routines ...................................... 8
- **assign and Fortran I/O** ...................................... 9
  - Alternative File Names ....................................... 9
  - File Structure Selection .................................... 10
    - Unblocked File Structure ................................ 13
    - `assign -s sbin` File Processing (not recommended) 14
    - `assign -s bin` File Processing .......................... 14
    - `assign -s u` File Processing ............................ 14
    - `text` File Structure ..................................... 15
    - `cos` or `blocked` File Structure ......................... 15
- **Buffer Specifications** ....................................... 17
  - Default Buffer Sizes ......................................... 19
  - Library Buffering ............................................ 19
  - System Cache .................................................. 20
Unbuffered I/O ................................................................. 20
Foreign File Format Specification ........................................ 21
Memory Resident Files ....................................................... 21
Fortran File Truncation ..................................................... 21
The Assign Environment File ............................................... 23
Local Assign Mode .......................................................... 23
  Example 1: Local assign mode .......................................... 23

Using FFIO [3] .................................................................. 25
Introduction to FFIO .......................................................... 25
Using Layered I/O ............................................................. 27
  I/O Layers ................................................................. 28
  Layered I/O Options ..................................................... 29
FFIO and Common Formats ............................................... 31
  Reading and Writing Text Files ........................................ 31
  Reading and Writing Unblocked Files ................................. 32
  Reading and Writing Fixed-length Records ......................... 32
  Reading and Writing Blocked Files .................................. 33
Enhancing Performance ..................................................... 33
  Buffer Size Considerations ............................................ 33
  Removing Blocking ....................................................... 33
    The syscall Layer ..................................................... 34
    The bufa and cachea Layers ........................................ 34
    The mr Layer ........................................................ 35
    The global Layer ...................................................... 35
    The cache Layer ....................................................... 35
Sample Programs ........................................................... 37
  Example 2: Unformatted direct mr with unblocked file ............ 37
  Example 3: Unformatted sequential mr with blocked file .......... 38
## Appendix A  FFIO Layer Reference

<table>
<thead>
<tr>
<th>Layer</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristics of Layers</td>
<td>42</td>
</tr>
<tr>
<td>The bufa Layer</td>
<td>43</td>
</tr>
<tr>
<td>The cache Layer</td>
<td>45</td>
</tr>
<tr>
<td>The cachea Layer</td>
<td>47</td>
</tr>
<tr>
<td>The cos Blocked Layer</td>
<td>48</td>
</tr>
<tr>
<td>The event Layer</td>
<td>50</td>
</tr>
<tr>
<td>The f77 Layer</td>
<td>51</td>
</tr>
<tr>
<td>The fd Layer</td>
<td>53</td>
</tr>
<tr>
<td>The global Layer</td>
<td>53</td>
</tr>
<tr>
<td>The ibm Layer</td>
<td>55</td>
</tr>
<tr>
<td>The mr Layer</td>
<td>58</td>
</tr>
<tr>
<td>The null Layer</td>
<td>61</td>
</tr>
<tr>
<td>The syscall Layer</td>
<td>62</td>
</tr>
<tr>
<td>The system Layer</td>
<td>63</td>
</tr>
<tr>
<td>The text Layer</td>
<td>63</td>
</tr>
<tr>
<td>The user and site Layers</td>
<td>65</td>
</tr>
<tr>
<td>The vms Layer</td>
<td>65</td>
</tr>
</tbody>
</table>

## Appendix B  Creating a user Layer

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Functions</td>
<td>69</td>
</tr>
<tr>
<td>The Operations Structure</td>
<td>70</td>
</tr>
<tr>
<td>FFIO and the stat Structure</td>
<td>71</td>
</tr>
<tr>
<td>user Layer Example</td>
<td>72</td>
</tr>
</tbody>
</table>

## Appendix C  Numeric File Conversion Routines

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion Overview</td>
<td>97</td>
</tr>
<tr>
<td>Transferring Data</td>
<td>98</td>
</tr>
<tr>
<td>Using fdcp to Transfer Files</td>
<td>98</td>
</tr>
<tr>
<td>Using ftp to Move Data between Systems</td>
<td>98</td>
</tr>
<tr>
<td>Data Item Conversion</td>
<td>98</td>
</tr>
</tbody>
</table>
This publication describes the Fortran input/output (I/O) techniques in support of the Programming Environment 5.0 releases. This manual is intended for Fortran, C, and C++ programmers who need general or advanced I/O information.

This preface describes how to access Cray documentation and error message explanations, interpret our typographical conventions, order Cray documentation, and contact us about this document.

Accessing Cray Documentation

Each software release package includes the CrayDoc documentation system, a collection of open-source software components that gives you fast, easy access to and the ability to search all Cray manuals, man pages, and glossary in HTML and/or PDF format from a web browser at the following locations:

- Locally, using the network path defined by your system administrator
- On the Cray public web site at:
  
  http://www.cray.com/craydoc/

All software release packages include a software release overview that provides information for users, user services, and system administrators about that release. An installation guide is also provided with each software release package. Release overviews and installation guides are supplied in HTML and PDF formats as well as in printed form. Most software release packages contain additional reference and task-oriented documentation, like this document, in HTML and/or PDF formats.

Man pages provide system and programming reference information. Each man page is referred to by its name followed by a number in parentheses:

manpagename (n)

where n is the man page section identifier:

1    User commands
2    System calls
3    Library routines
4    Devices (special files) and Protocols
5 File formats
7 Miscellaneous information
8 Administrator commands

Access man pages in any of these ways:

- Enter the `man` command to view individual man pages in ASCII format; for example:
  ```
  man ftn
  ```
  To print individual man pages in ASCII format, enter, for example:
  ```
  man ftn | col -b | lpr
  ```

- Use a web browser with the CrayDoc system to view, search, and print individual man pages in HTML format.

- Use Adobe Acrobat Reader with the CrayDoc system to view, search, and print from `collections` of formatted man pages provided in PDF format.

If more than one topic appears on a page, the man page has one primary name (`grep`, for example) and one or more secondary names (`egrep`, for example). Access the ASCII or HTML man page using either name; for example:

- Enter the command `man grep` or `man egrep`
- Search in the CrayDoc system for `grep` or `egrep`

### Error Message Explanations

Access explanations of error messages by entering the `explain msgid` command, where `msgid` is the message ID string in the error message. For more information, see the `explain(1)` man page.
Typographical Conventions

The following conventions are used throughout this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.</td>
</tr>
<tr>
<td>variable</td>
<td>Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace filename with the name datafile in your program. It also denotes a word or concept being defined.</td>
</tr>
<tr>
<td>user input</td>
<td>This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.</td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate that a preceding element can be repeated.</td>
</tr>
</tbody>
</table>

Ordering Documentation

To order software documentation, contact the Cray Software Distribution Center in any of the following ways:

E-mail:
orderidsk@cray.com

Web:
http://www.cray.com/craydoc/

Click on the Cray Publications Order Form link.

Telephone (inside U.S., Canada):
1-800-284-2729 (BUG CRAY), then 605–9100

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–651–605–9100
Fax:
+1–651–605–9001

Mail:
Software Distribution Center
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
swpubs@cray.com

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–715–726–4993 (Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA
This publication describes advanced Fortran input/output (I/O) techniques for use on Cray X1 systems. The information in this manual focuses on three general areas:

- The `assign` environment, which enables you to alter many details of a Fortran file connection including device residency, alternate files name, file structure, buffer sizes, and data conversion properties.

- The flexible file I/O (FFIO) system, which, when used in the context of the `assign` environment, enables you to use sophisticated I/O techniques to optimize program performance and quickly tune I/O strategies without modifying source code, recompiling, or relinking.

- The numeric conversion utilities, which enable you to move data to and from other computer systems.

The reader should be familiar with the information presented in the following Cray publications:

- *Fortran Language Reference Manual, Volume 2*, especially Chapters 1 and 2, which cover the fundamentals of Fortran I/O processing and editing techniques

- *Fortran Language Reference Manual, Volume 3*, especially Chapter 4, which discusses I/O-related Fortran extensions

- The `assign(1)`, `assign(3f)`, and `ffassign(3f)` man pages

- The `intro_ffio(1)` man page, which describes the FFIO system and performance options available with the FFIO layers

Additional information that may be helpful can be found in these documents:

- *Fortran Language Reference Manual, Volume 1*

- *Cray Fortran Compiler Commands and Directives Reference Manual*

- *Optimizing Applications on the Cray X1 System*

- *Man Page Collection: Fortran and C/C++ Intrinsic Procedures*
Using the Assign Environment [2]

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. These details comprise the *assign environment*.

In addition, Cray X1 systems support *flexible file I/O* (FFIO), which uses layered I/O to implement sophisticated I/O strategies. When used in the context of the assign environment, FFIO enables you to implement different I/O techniques and realize significant improvements in I/O performance without modifying source code.

This chapter describes the `assign(1)` command and the `assign(3f)` library routine, which together define the assign environment.

The FFIO system is described in Chapter 3, page 25.

The `ffassign(3c)` command provides an interface to assign processing from C/C++. See the `ffassign(3c)` man page for details about its use.

### 2.1 assign Basics

The `assign` command information is stored in the assign environment file `.assign` or in a shell environment variable. To begin using the assign environment to control a program’s I/O behavior, follow these steps.

1. Set the `FILENV` environment variable to the desired path:
   ```
   set FILENV environment_file
   ```

2. Run the `assign` command to define the current assign environment:
   ```
   assign arguments assign_object
   ```
   For example:
   ```
   assign -F f77 g:su
   ```

3. Run your program:
   ```
   ./a.out arguments
   ```
4. If you are not satisfied with the I/O performance observed during program execution, return to step 2, use the assign command to adjust the assign environment, and try again.

The assign(l) command passes information to Fortran open statements and to the ffopen(3c) routine to identify the following elements:

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

The assign object is the file name, file name pattern, unit number, or type of I/O open request to which the assign environment applies. When the unit or file is opened from Fortran, the environment defined by the assign command is used to establish the properties of the connection.

2.1.1 Assign Objects and Open Processing

The I/O library routines apply options to a file connection for all related assign objects.

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 library I/O routines opens a file, it uses the specified assign environment options for any assign objects that apply to the open request. Any of the following assign objects or categories might apply to a given open request:

- \( g: \text{all} \) options apply to any open request.
- \( g: \text{su}, g: \text{du}, g: \text{sf}, g: \text{df}, \) and \( g: \text{ff} \) all apply to types of open requests. These equate to sequential unformatted, direct unformatted, sequential formatted, direct formatted, or ffopen, respectively.
- \( u: \text{unit\_number} \) applies whenever \( \text{unit\_number} \) is opened.
- \( p: \text{pattern} \) applies whenever a file whose name matches \( \text{pattern} \) is opened. The assign environment can contain only one \( p: \text{assign\_object} \) that matches the current open file. The exception is that the \( p: \%\text{pattern} \) (which uses the \% wildcard character) is silently ignored if a more specific \text{pattern} also matches the current file name being opened.
Using the Assign Environment [2]

• `£: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.

2.1.2 The assign Command

Here is the syntax for the `assign` command:

```
       [-V] [-W setting] [-Y setting] [-Z setting] assign_object
```

The following specifications cannot be used with any other options:

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

A summary of the `assign` command options follows. For details, see the `assign(1)` and `intro_ffio(3f)` man pages.

Here 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_objects` are removed.

- **-V**
  Views attributes for `assign_object`. If `assign_object` is not specified, all currently assigned attributes for all `assign_objects` are printed.
Here are the assign command attribute options:

-a actualfile  The file= specifier or the actual file name.

-b bs         Library buffer size in 4096-byte (512-word) blocks.

-f forstd      Specifies compatibility with a Fortran standard, where forstd is either 77 for Cray Fortran 77, 90 for Cray Fortran 90 or 95, irixf77 for SGI Fortran 77, or irixf90 for SGI Fortran 90.

-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.

-s ft         File type. Enter text, cos, blocked, unblocked, u, sbin, or bin for ft. The default is text.

-t            Temporary file.

-u buflen     Buffer count. Specifies the number of buffers to be allocated for a file.

-y setting    Suppresses repeat counts in list-directed output. setting can be either on or off. The default setting is off.

-B setting    Activates or suppresses the passing of the O_DIRECT flag to the open(2) system call. Enter either on or off for setting. This is an important feature for I/O optimization; if this is on, it enables reads and writes directly to and from the user program buffer.

-C charcon    Character set conversion information. Enter ascii, or ebcidic for charcon. If you specify the -C option, you must also specify the -F option.

-D fildes     Specifies a connection to a standard file. Enter stdin, stdout, or stderr for fildes.

-F spec [, specs] Flexible file I/O (FFIO) specification. See the assign(1) man page for details about allowed values for spec and for details about hardware
platform support. See the intro_ffio(3f) man page for details about specifying the FFI0 layers.

-N numcon
Foreign numeric conversion specification. See the assign(l) man page for details about allowed values for numcon and for details about hardware platform support.

-S setting
Suppresses use of a comma as a separator in list-directed output. Enter either on or off for setting. The default setting is off.

-T setting
Activates or suppresses truncation after write for sequential Fortran files. Enter either on or off for setting.

-U setting
Produces a non-UNICOS form of list-directed output. This is a global setting that sets the value for the -y, -S, and -W options. Enter either on or off for setting. The default setting is off.

-W setting
Suppresses compressed width in list-directed output. Enter either on or off for setting. The default setting is off.

-Y setting
Skips unmatched namelist groups in a namelist input record. Enter either on or off for setting. The default setting is off.

-Z setting
Recognizes –0.0 for IEEE floating point systems and writes the minus sign for edit-directed, list-directed, and namelist output. Enter either on or off for setting. The default setting is on.

assign_object
Specify 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, or ff (for example, g:ff for ffopen(3C))
- p:pattern (for example, p:file%)
2.1.3 Assign Library Routines

The `assign3f`, `asnunit3f`, `asnfile3f`, and `asnrm3f` 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:

```fortran
    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 `pxfsystem` 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.

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.
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 asmunit(iun,’-i -n 2’,ier)

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

call asnrm(ier)

2.2 assign and Fortran I/O

Assign processing lets you tune file connections. This sections describes several areas of assign command usage and provide examples of each use.

2.2.1 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 different directories without changing the FILE= specifier on an OPEN statement.

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

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

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

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

Unit 1 is connected to file /tmp/mydir/tmpfile. Without the -a attribute, unit 1 would be connected to file fort.1.
When the -a attribute is associated with a file, any Fortran open that is set to connect to the file causes a connection to the actual file name. An assign command of the following form causes a connection to file FILENV/joe:

assign -a FILENV/joe ftfile

This is true when the following statement is executed in a program:

OPEN(IUN,FILE='ftfile')

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 that 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.

2.2.2 File Structure Selection

A file structure defines the way records are delimited and how the end-of-file is represented.
The `assign` command supports two mutually exclusive file structure options. Use the `-s` option to select a structure explicitly, or use the `-F` option to select a structure using an FFIO layer.

As a general rule, the `-F` option is more flexible than the `-s` option; it allows nested file structures, buffer size specifications, and support for file structures that are not available through the `-s` option. You will also realize better I/O performance by using the `-F` option and FFIO layers.

The remainder of this section covers the `-s` option.

Fortran I/O uses four different file structures: `f77` blocked structure, `text` structure, unblocked structure, and COS blocked structure. By default, the `f77` blocked structure is used unless a file structure is selected at open time. If an alternative file structure is needed, the user can select a file structure by using the `-s` or `-F` option on the `assign` command.

The `-s` and `-F` options are mutually exclusive. The following list summarizes how to select the different file structures with different options to the `assign` command:

<table>
<thead>
<tr>
<th>Structure</th>
<th><code>assign</code> command</th>
</tr>
</thead>
<tbody>
<tr>
<td>COS blocked</td>
<td><code>assign -F cos</code></td>
</tr>
<tr>
<td></td>
<td><code>assign -s cos</code></td>
</tr>
<tr>
<td>text</td>
<td><code>assign -F text</code></td>
</tr>
<tr>
<td></td>
<td><code>assign -s text</code></td>
</tr>
<tr>
<td>unblocked</td>
<td><code>assign -F system</code></td>
</tr>
<tr>
<td></td>
<td><code>assign -s unblocked</code></td>
</tr>
<tr>
<td></td>
<td><code>assign -s u</code></td>
</tr>
<tr>
<td>F77 blocked</td>
<td><code>assign -F f77</code></td>
</tr>
</tbody>
</table>

The following examples address file structure selection:

- To select an unblocked file structure for a sequential unformatted file:
  ```fortran
  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:
  ```fortran
  ```
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:
  \texttt{assign -s cos u:10}

The full set of options allowed with the \texttt{assign -s} command are as follows:
- \texttt{bin} (not recommended)
- \texttt{blocked}
- \texttt{cos}
- \texttt{sbin}
- \texttt{text}
- \texttt{u}
- \texttt{unblocked}

Table 1 summarizes the Fortran access methods and options.

<table>
<thead>
<tr>
<th>Access and form</th>
<th>assign -s ft defaults</th>
<th>assign -s ft options</th>
</tr>
</thead>
</table>
| Sequential unformatted BUFFER IN and BUFFER OUT | \texttt{blocked / cos / f77} | \texttt{bin}
  \texttt{sbin}
  \texttt{u}
  \texttt{unblocked} |
| Direct unformatted | \texttt{unblocked} | \texttt{bin}
  \texttt{sbin}
  \texttt{u}
  \texttt{unblocked} |
| Sequential formatted | \texttt{text} | \texttt{blocked}
  \texttt{cos}
  \texttt{sbin/text} |
| Direct formatted | \texttt{text} | \texttt{sbin/text} |
2.2.2.1 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.

Do not reposition a file with unblocked file structure with a BACKSPACE statement. 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 has 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(l) 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 end-of-file record in the file. The following unblocked data file structure specifications are available:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign -s unblocked</td>
<td>Library-buffered</td>
</tr>
<tr>
<td>assign -F system</td>
<td>No library buffering</td>
</tr>
<tr>
<td>assign -s u</td>
<td>No library buffering</td>
</tr>
<tr>
<td>assign -s sbin</td>
<td>Buffering that is compatible with standard I/O; for example, both library and system buffering.</td>
</tr>
</tbody>
</table>

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.

For more information on buffering, see Section 2.2.3.

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 UNICOS/mp contains records that are the same length. The end-of-file record is recognized in sequential-access files.

2.2.2.2 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.

Note: Use of assign -s sbin is discouraged because of poor I/O performance. If you cannot use an FFI0 layer, use assign -s text for formatted files and assign -s unblocked for unformatted files.

2.2.2.3 assign -s bin File Processing

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. 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.

2.2.2.4 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.

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.
2.2.2.5 **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. 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.

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. You can use a `BACKSPACE` statement to reposition a file with this structure.

---

2.2.2.6 **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.

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.

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 illustration and table is a representation of the structure of a BCW.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>0–3</td>
<td>Type of control word; 0 for BCW</td>
</tr>
<tr>
<td>bdf</td>
<td>11</td>
<td>Bad Data flag (1-bit).</td>
</tr>
<tr>
<td>bn</td>
<td>31–54</td>
<td>Block number (modulo 2^24).</td>
</tr>
<tr>
<td>fwi</td>
<td>55–63</td>
<td>Forward index; the number of words to next control word.</td>
</tr>
</tbody>
</table>

The following illustration and table is a representation of the structure of an RCW.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>0–3</td>
<td>Type of control word; 108 for EOR, 168 for EOF, and 178 for EOD.</td>
</tr>
<tr>
<td>ubc</td>
<td>4–9</td>
<td>Unused bit count; number of unused low-order bits in last word of previous record.</td>
</tr>
</tbody>
</table>
### Field Bits Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran</td>
<td>10</td>
<td>Transparent record field (unused).</td>
</tr>
<tr>
<td>bdf</td>
<td>11</td>
<td>Bad data flag (unused).</td>
</tr>
<tr>
<td>srs</td>
<td>12</td>
<td>Skip remainder of sector (unused).</td>
</tr>
<tr>
<td>pfi</td>
<td>20–39</td>
<td>Previous file index; offset modulo $2^{20}$ to the block where the current file starts (as defined by the last EOF).</td>
</tr>
<tr>
<td>pri</td>
<td>40–54</td>
<td>Previous record index; offset modulo $2^{15}$ to the block where the current record starts.</td>
</tr>
<tr>
<td>fwi</td>
<td>55–63</td>
<td>Forward index; the number of words to next control word.</td>
</tr>
</tbody>
</table>

### 2.2.3 Buffer Specifications

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.

- Many data file structures such as cos 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 remove the control words before passing the data on to the user (called 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 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 I/O devices and the processors in parallel often improves performance; therefore, it is useful to keep processors busy while data is being moved. To do this when writing, data can be transferred to the buffer at memory-to-memory copy speed. Use an asynchronous I/O request. 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 called read-ahead 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 use of a cache.

- When direct I/O is enabled (assign -B on), data is staged in the system buffer cache. While this can yield improved performance, it also means that performance is affected by program competition for system buffer cache. To minimize this effect, avoid public caches when possible.

- In many cases, the best asynchronous I/O performance can be realized by using the FFIO cachea layer (assign -F cachea). This layer supports read-ahead, write-behind, and improved cache reuse.

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 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 the assign(1) man page.

The following examples of buffer size specification illustrate 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 buffer_size u:buffer_count
  ```

- 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
  ```
2.2.3.1 Default Buffer Sizes

The Fortran I/O library automatically selects default buffer sizes according to file access type as shown in Table 2. You can override the defaults by using the assign(1) command. The following subsections describe the default buffer sizes on various systems.

Note: One block is 4,096 bytes on UNICOS/mp systems.

The default buffer sizes are as follows:

Table 2. Default Buffer Sizes for Fortran I/O Library Routines

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Default Buffer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential formatted</td>
<td>16 blocks (65,536 bytes)</td>
</tr>
<tr>
<td>Sequential unformatted</td>
<td>16 blocks (65,536 bytes)</td>
</tr>
<tr>
<td>Direct formatted</td>
<td>The smaller of:</td>
</tr>
<tr>
<td></td>
<td>• The record length in bytes + 1</td>
</tr>
<tr>
<td></td>
<td>• 16 blocks (65,536 bytes)</td>
</tr>
<tr>
<td>Direct unformatted</td>
<td>The larger of:</td>
</tr>
<tr>
<td></td>
<td>• The record length</td>
</tr>
<tr>
<td></td>
<td>• 16 blocks (65,536 bytes)</td>
</tr>
</tbody>
</table>

The maximum default buffer size is 100 blocks. Four buffers of this size are allocated. For more information, see the description of the cachea layer in the intro_ffio(3F) man page.

2.2.3.2 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 command to determine the type of processing that should be used on the file. Buffers are an integral part of the processing.

If the file is assigned with one of the following assign(1) options, library buffering is used:

- `s blocked`
- `-F spec` (buffering as defined by `spec`)
The \texttt{-F} option specifies flexible file I/O (FFIO), which uses library buffering if the specifications selected include a need for buffering. In some cases, more than one set of buffers might be used in processing a file. For example, the \texttt{-F bufa, 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 blank compression. In other cases (for example, \texttt{-F system}), no library buffering occurs.

### 2.2.3.3 System Cache

The operating system uses a set of buffers in kernel memory for I/O operations. These are collectively called the \textit{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.

The following \texttt{assign(1)} command options can be expected to use system cache:

\begin{itemize}
  \item \texttt{-s sbin}
  \item \texttt{-F spec (FFIO, depends on spec)}
\end{itemize}

For the \texttt{assign -F cachea} command, a library buffer ensures that the actual system calls are well formed and the system buffer cache is bypassed. This is not true for the \texttt{assign -s u} option. If you plan to bypass the system cache, all requests go through the cache except those that are well formed.

### 2.2.3.4 Unbuffered I/O

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

\texttt{assign -s u ...}

Use the \texttt{assign} command to bypass library buffering and the 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.
2.2.4 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:

```fortran
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 ebc dic fgnfile
```

2.2.5 Memory Resident Files

The `assign -F mr` command specifies that a file will be memory resident. Because the `mr` flexible file I/O layer does not define a record-based file structure, it must be nested beneath a file structure layer when record blocking is needed.

For example, if unit 2 is a sequential unformatted file that is to be memory resident, the following Fortran statements connect the unit:

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

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

2.2.6 Fortran File Truncation

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

Figure 1 summarizes the available access methods and the default buffer sizes.

<table>
<thead>
<tr>
<th>Access method assign option</th>
<th>Blocked</th>
<th>Unblocked</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocked</td>
<td>Unblocked</td>
</tr>
<tr>
<td>FORMATTED SEQ. I/O WRITE(9,20) PRINT</td>
<td>Valid</td>
<td>Valid Default</td>
</tr>
<tr>
<td>FORMATTED DIRECT I/O WRITE(9,20,REC=)</td>
<td>Valid Default</td>
<td>Valid</td>
</tr>
<tr>
<td>UNFORMATTED SEQ. I/O WRITE(9)</td>
<td>Valid Default</td>
<td>Valid</td>
</tr>
<tr>
<td>UNFORMATTED DIRECT I/O WRITE(9,REC=)</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>BUFFER IN/BUFFER OUT</td>
<td>Valid Default</td>
<td>Valid</td>
</tr>
</tbody>
</table>

| Control words | Yes | Yes | NEWLINE | No | No | No |
| Library buffering | Yes | Yes | Yes | No | Yes | Yes |
| System cached | Yes | No | Yes | Not† | Not† | Varies |
| BACKSPACE | Yes | Yes | Yes | No | No | No |
| Record size | Any | Any | Any | Any | 8*n | Any |
| Default library buffer size* | 16 | 48 | 16 | None | 16 | 16 |

† Cached if not well-formed
†† No guarantee when physical size not 512 words
* In units of 4096 bytes, unless otherwise specified

Figure 1. Access methods and default buffer sizes
2.3 The Assign Environment File

The assign command information is stored in the assign environment file .assign or in a shell environment variable. The location of the current assign environment file must be provided by setting the FILENV environment variable to the desired path.

The format of the .assign environment file is subject to change with each release.

2.4 Local Assign Mode

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 can 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 1: Local assign mode

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

C Switch to local assign environment
CALL ASNCTL('LOCAL',1,IER)
IUN = 11
C Assign the unblocked file structure
CALL ASNUNIT(IUN,'-I -s unblocked',IER)
C Open unit 11
OPEN(IUN,FOR='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,FOR=’UNFORMATTED’)
This chapter provides an overview of the capabilities of the flexible file I/O (FFIO) system and describes how to use FFIO with common file structures to enhance code performance without changing source code.

Flexible file I/O—sometimes called layered I/O—is used to perform many I/O-related tasks. For details about each individual I/O layer, see Appendix A, page 41.

3.1 Introduction to FFIO

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. I/O can be the slowest part of a computational process, and the speed of I/O access methods varies depending on computational processes.

Figure 2 illustrates the typical flow of data from the user’s variables to and from the I/O device.

![Figure 2. Typical Data Flow](image)

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. It is also important to realize that the actual I/O path can skip one or more steps in this process, depending on the I/O features used at a given point in a given program.

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 or aborted. 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. You can specify a comma-separated list of one or more processing steps by using the `assign -F` command:

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

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

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

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

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

The FFIO system has an internal model of the world of data, which it maps to any given actual logical file type. Four of these concepts are basic to understanding the inner workings of the layers.
Concept | Definition
---|---
Data | Data is a stream of bits.
Record marks | End-of-record (EOR) marks are boundaries between logical records.
File marks | End-of-file (EOF) marks 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 and/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
- Formatted sequential
- Namelist
- List-directed

### 3.2 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 f77 on UNICOS/mp systems. The -F f77 option provides the same structure. The FFIO system provides detailed control over I/O processing requests. However, to effectively use the f77 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:

```fortran
assign -F system
```

The `system` layer is a generic system interface that chooses an appropriate layer for your file. If the file is on the RAID, 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.

If you want your file to be F77 blocked (the default blocking for Fortran unformatted I/O on UNICOS/mp systems), you can specify:

```fortran
assign -F f77
```

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

```fortran
assign -F cos
```

**Note:** In all `assign -F` specifications, the `system` layer is the implied last layer. The above example is functionally identical to `assign -F cos,system`.

These two specs request that each `WRITE` request first be blocked (blocking adds control words to the data in the file to delimit records). The `f77` 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 `f77` layer, where it is deblocked. The deblocked data is then presented to the user.

### 3.2.1 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 `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.
Table 3 defines the classes you can specify for the `spec` argument to the `assign` `-F` option:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufa</td>
<td>Asynchronous buffering layer</td>
</tr>
<tr>
<td>cache</td>
<td>Memory cached I/O</td>
</tr>
<tr>
<td>cachea</td>
<td>Asynchronous memory cached I/O</td>
</tr>
<tr>
<td>cos or blocked</td>
<td>COS blocking</td>
</tr>
<tr>
<td>event</td>
<td>I/O monitoring layer</td>
</tr>
<tr>
<td>f77</td>
<td>Record blocking common to most UNIX Fortran</td>
</tr>
<tr>
<td>fd</td>
<td>File descriptor open</td>
</tr>
<tr>
<td>global</td>
<td>Distributed cache layer for MPI, SHMEM, OpenMP, and Co-array Fortran</td>
</tr>
<tr>
<td>ibm</td>
<td>IBM file formats</td>
</tr>
<tr>
<td>mr</td>
<td>Memory-resident file handlers</td>
</tr>
<tr>
<td>null</td>
<td>Syntactic convenience for users (does nothing)</td>
</tr>
<tr>
<td>site</td>
<td>Site-specific layer</td>
</tr>
<tr>
<td>syscall</td>
<td>System call I/O</td>
</tr>
<tr>
<td>system</td>
<td>Generic system interface</td>
</tr>
<tr>
<td>text</td>
<td>Newline separated record formats</td>
</tr>
<tr>
<td>user</td>
<td>User-defined layer</td>
</tr>
<tr>
<td>vms</td>
<td>VAX/VMS file formats</td>
</tr>
</tbody>
</table>

### 3.2.2 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 Table 3. Each layer 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 specs are identical:
  
  ```
  IbVm.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 specs are identical:
  
  ```
  cos...:40, cos..:40
  cos::40
  ```

  In this example, *opt1*, *opt2*, *num1*, and *num2* can assume default values.

- 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 *vms* layer on UNICOS systems and set the initial allocation to 100 blocks:

```
-F vms:100
-F vms.:100
-F vms..:100
```
When possible, the default settings of the layers are set so that optional fields are seldom needed.

3.3 FFIO and Common Formats

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

3.3.1 Reading and Writing Text Files

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

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 processor time, but it lets you use memory-resident files, which can reduce or eliminate I/O wait time.

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

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:

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

If the name of the script is to.text, you can invoke it as follows:
% to.text cos data_cos data_text

3.3.2 Reading and Writing Unblocked Files

The simplest data file format is the 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 processor and system overhead.

The FFIO package provides several layers designed specifically to handle a binary stream of data. These layers are syscall, mr, bufa, cache, cachea, and global. 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 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.

3.3.3 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 syscall, syscall, cache, cachea, global, and mr. 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.
3.3.4 Reading and Writing Blocked Files

The $f77$ blocking format is the default file structure for all Fortran sequential unformatted files. The $f77$ layer is provided to handle these files. It provides for $f77$ blocked files on the RAID and it supports multiframe $f77$ blocked data sets.

The $f77$ layer must be specified for $f77$ blocked files. If $f77$ is not the default file structure, or if you specify another layer, such as $mr$, you may have to specify a $f77$ layer to get $f77$ blocking.

3.4 Enhancing Performance

FFIO can be used to enhance performance in a program without changing or recompiling the source code. This section describes some basic techniques used to optimize I/O performance. Additional optimization options are discussed in Appendix A, page 41.

3.4.1 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

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.

3.4.2 Removing Blocking

I/O optimization usually consists of reducing overhead. One part of the overhead in doing I/O is the processor 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:
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.

3.4.2.1 The syscall Layer

The syscall layer offers a simple, direct system interface with a minimum of system and library overhead. If requests are larger than approximately 64 K, this method can be appropriate. However, the I/O requests must meet certain criteria in order to be efficient:

- The buffers must be aligned on physical page boundaries.
- The file addresses must be multiples of 512 bytes.
- The I/O sizes must be 4096 bytes or larger.
- The I/O sizes must not exceed the maximum I/O size for the system.

3.4.2.2 The bufa and cachea Layers

The bufa and cachea layers permit efficient file processing. Both layers provide asynchronous buffering managed by the library, and the cachea layer allows recently accessed parts of a file to be cached in memory.

The number of buffers and the size of each buffer are 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 depends on the st_blksize field returned from a stat(2) system call of the file; if this return value is 0, the default is 8 for all files. The nbufs argument specifies the number of buffers to use. bufa defaults to 2 buffers, while cachea defaults to 16 buffers.
3.4.2.3 The mr Layer

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 a file, or part of a file, that can reside in memory.

The mr layer features both scr and save mode, and it directs overflow to the next lower layer automatically.

The assign -F command specifies the entire set of processing steps that are performed when I/O is requested. If a file is blocked, you must specify the appropriate layer for the handling of block and record control words as in the following examples:

assign -F f77,mr u:1
assign -F cos,mr fort.1

Section 3.5, page 37 contains several mr program examples.

3.4.2.4 The global Layer

The global layer is a caching layer that distributes data across all multiple SHMEM or MPI processes. Open and close operations require participation by all processes that access the file; all other operations are performed independently by one or more processes. File positions can be private to a process or global to all processes.

You can specify both the cache size and the number of cache pages to use. Since this layer is used by parallel processes, the actual number of cache pages used is the number specified times the number of processes.

3.4.2.5 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 16. 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 RAID stripe 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:

```fortran
integer VECTSIZE, NUMCHUNKS, CHUNKSIZE
parameter(VECTSIZE=1000*512)
parameter(NUMCHUNKS=100)
parameter(CHUNKSIZE=VECTSIZE/NUMCHUNKS)
read a(CHUNKSIZE), sum(CHUNKSIZE)
on(11,access='direct',recl=CHUNKSIZE*8)
call asunit (2, 's unblocked', ier)
on (2,form='unformatted')
d 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 MB of memory are allocated for buffers for unit 11, 10 cache pages should be used, each of the following size:

$4MB/10 = 400000 \text{ bytes} = 97 \text{ blocks}$
Make the buffer size an even multiple of the record length of 409600 bytes by rounding it up to 100 blocks (= 409600 bytes), then use the following assign command:

\texttt{assign -F cache:100:10 u:11}

### 3.5 Sample Programs

The following examples illustrate the use of the \texttt{mr} layers.

**Example 2: Unformatted direct mr with unblocked file**

In the following example, batch job \texttt{ex8} contains a program that uses unformatted direct-access I/O with an \texttt{mr} layer:

```bash
#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
  end do 100
  close(1)
end
EOF
```
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 RAID. Because the `scr` option is specified, the file is not saved when `fort.1` is closed.

**Example 3: Unformatted sequential `mr` with blocked file**

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

```fortran
program example4a
  integer r(512)
  data r/512*1.0/
  C Reset assign environment, then assign file without FFIO 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.ierr2.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)
  goto200
end
```

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

```fortran
assign -R
assign -a /tmp/file1 -F f77,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 instructions 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.
This appendix provides details about each of the following FFIO layers:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufa</td>
<td>Library-managed asynchronous buffering</td>
</tr>
<tr>
<td>cache</td>
<td>Memory-cached layer</td>
</tr>
<tr>
<td>cachea</td>
<td>Asynchronous memory-cached layer</td>
</tr>
<tr>
<td>cos or blocked</td>
<td>COS blocking layer</td>
</tr>
<tr>
<td>event</td>
<td>I/O monitoring layer</td>
</tr>
<tr>
<td>f77</td>
<td>Common UNIX Fortran record blocking</td>
</tr>
<tr>
<td>fd</td>
<td>File descriptor open layer</td>
</tr>
<tr>
<td>global</td>
<td>Distributed I/O for MPI, SHMEM, OpenMP, and Co-array Fortran programs</td>
</tr>
<tr>
<td>ibm</td>
<td>IBM file formats</td>
</tr>
<tr>
<td>mr</td>
<td>Memory-resident file handlers</td>
</tr>
<tr>
<td>null</td>
<td>Syntactic convenience for users</td>
</tr>
<tr>
<td>site</td>
<td>User-defined site-specific layer</td>
</tr>
<tr>
<td>syscall</td>
<td>System call I/O</td>
</tr>
<tr>
<td>system</td>
<td>Generic system layer</td>
</tr>
<tr>
<td>text</td>
<td>Newline-separated record formats</td>
</tr>
<tr>
<td>user</td>
<td>User-defined layer</td>
</tr>
<tr>
<td>vms</td>
<td>VAX/VMS file formats</td>
</tr>
</tbody>
</table>

Section A.1 describes how to interpret the information presented in the remaining sections of this appendix. See the `intro_ffio(3)` man page for more details about FFIO layers.
A.1 Characteristics of Layers

In the descriptions of the layers that follow, the Data Manipulation tables use the following categories of characteristics:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>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 process data in units of 6-bit characters in which any operation that is not a multiple of 6 bits results in an error.</td>
</tr>
<tr>
<td>Data model</td>
<td>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).</td>
</tr>
<tr>
<td>Truncate on write</td>
<td>Indicates whether the layer forces an implied EOD on every write operation (EOD implies truncation).</td>
</tr>
<tr>
<td>Implementation</td>
<td>Describes the internal routines that are used to implement the layer. The X-records type under Implementation Strategy (if used in the tables) 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.</td>
</tr>
</tbody>
</table>

In the descriptions of the layers, the Supported Operations tables use the following categories:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Lists the operations that apply to that particular layer. The following is a list of supported operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>ffclose</td>
</tr>
<tr>
<td>ffread</td>
<td>ffflush</td>
</tr>
<tr>
<td>ffreada</td>
<td>ffweof</td>
</tr>
<tr>
<td>ffreadc</td>
<td>ffweod</td>
</tr>
<tr>
<td>(ffwrite</td>
<td>ffseek</td>
</tr>
<tr>
<td>ffwritea</td>
<td>ffpos</td>
</tr>
<tr>
<td>ffwritec</td>
<td>ffbksp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Support</th>
<th>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.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Used</th>
<th>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.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Comments</th>
<th>Describes the function or support of the layer’s function.</th>
</tr>
</thead>
</table>

On many layers, you can also specify the numeric parameters by using a keyword. This functionality is available if your application is linked with CrayLibs 3.0 or later release.

### A.2 The bufa Layer

The bufa layer provides library-managed asynchronous buffering. It is optimized to perform sequential I/O using adaptive I/O techniques, meaning the bufa layer transforms READ and WRITE requests into read-ahead and write-behind requests. This can minimize I/O wait time and 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:
The `bufa[.bufsize=num1][.num_buffers=num2]` argument specifies the size, in 4096-byte blocks, of each buffer. The default buffer size depends on the device on which your file is located. The maximum allowed value on UNICOS/mp systems is 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 to be used. The default is 2.

### Table 4. Data Manipulation: bufa Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>Stream</td>
<td>No</td>
</tr>
<tr>
<td>8 bits</td>
<td>Stream</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 5. Supported Operations: bufa Layer

<table>
<thead>
<tr>
<th>Supported operations</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ffopen</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffread</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffrea</code></td>
<td>Yes, Always synchronous</td>
</tr>
<tr>
<td><code>ffreadc</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>fwritea</code></td>
<td>Yes, Always synchronous</td>
</tr>
<tr>
<td><code>fwritec</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffclose</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffflush</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffweof</code></td>
<td>Passed through</td>
</tr>
<tr>
<td><code>ffweod</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffseek</code></td>
<td>Yes, Only if supported by the underlying layer</td>
</tr>
</tbody>
</table>
## A.3 The cache Layer

The cache layer improves nonsequential I/O by dividing files into cache page-sized sections, then keeping whichever pages are currently being accessed in main memory. This can significantly improve data reuse, with appropriately configured buffers, and can also reduce the number of low-level I/O requests for random access.

When used as the last layer above the system or syscall layer, the cache layer supports the assign -B option to enable or disable direct I/O.

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 syntax is as follows:

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

The `type` argument can be `mem`. `mem` directs that cache pages reside in main memory. `num1` specifies the size, in 4096-byte blocks, of each cache page buffer. The default is 16. The maximum allowed value is 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 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 6. Data Manipulation: cache Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>Stream (mimics UNICOS/mp system calls)</td>
<td>No</td>
</tr>
<tr>
<td>8 bit</td>
<td>Stream</td>
<td>No</td>
</tr>
<tr>
<td>512 words</td>
<td>Stream</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 7. Supported Operations: cache Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>Yes</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>Yes</td>
</tr>
<tr>
<td>ffwritec</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffweof</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td></td>
<td>Yes Requires underlying interface to be a stream</td>
</tr>
<tr>
<td>ffpow</td>
<td>Yes</td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td></td>
<td>NA</td>
</tr>
</tbody>
</table>
A.4 The cachea Layer

The cachea layer is similar to the cache layer in that it improves data reuse and nonsequential I/O by dividing files into cache page-sized sections, then keeping whichever pages are currently being accessed in main memory. However, like the bufa layer, it also applies adaptive I/O techniques, transforming READ and WRITE operations into read-ahead and write-behinds. Furthermore, unlike the bufa layer, there can be multiple threads (I/O chains) of read-aheads and write-behinds, depending on how the file is being accessed.

As a result, 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.

When used as the last layer above the system or syscall layer, the cachea layer supports the assign -B option to enable or disable direct I/O.

The syntax is as follows:

```
cachea[type]:[num1]:[num2]:[num3]
```

The keyword syntax is as follows:

```
cachea[type].page_size=num1].num_pages=num2] [.max_lead=num3]
```

- **type**
  - Directs that cache pages reside in memory (mem).
- **num1**
  - Specifies the size, in 4096-byte blocks, of each cache page buffer. Default is 16. The maximum allowed value is 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.
Table 8. Data Manipulation: cachea Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>Stream (mimics UNICOS/mp system calls)</td>
<td>No</td>
</tr>
<tr>
<td>8 bit</td>
<td>Stream (mimics UNICOS/mp system calls)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 9. Supported Operations: cachea Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported operations</th>
<th>Comments</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>fwrite</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>fwritea</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>fwritec</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffweof</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td></td>
<td>Requires that the underlying interface be a stream</td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td></td>
<td>NA</td>
</tr>
</tbody>
</table>

A.5 The cos Blocked Layer

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

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

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

If not specified, the default buffer size is the larger of the following: the large I/O size, the preferred I/O block size (see the `stat(2)` man page for details), or 48 blocks. See the `intro_ffio(3F)` man page for more details.

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

Table 10. Data Manipulation: `cos` Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
<th>Implementation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>Records with multi-EOF capability</td>
<td>Yes</td>
<td><code>cos</code> specific</td>
</tr>
</tbody>
</table>

Table 11. Supported Operations: `cos` Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ffopen</code></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffread</code></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffreada</code></td>
<td>Yes</td>
<td>Always synchronous</td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffreadc</code></td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td><code>ffwrite</code></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffwritea</code></td>
<td>Yes</td>
<td>Always synchronous</td>
<td>Yes</td>
</tr>
<tr>
<td><code>ffwritec</code></td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td><code>ffclose</code></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Operation</td>
<td>Supported</td>
<td>Comments</td>
<td>Required of next lower level?</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>fflush</td>
<td>Yes</td>
<td>No-op</td>
<td>Yes</td>
</tr>
<tr>
<td>ffw.eof</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>ffw.od</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ff.seek</td>
<td>Yes</td>
<td>Minimal support (see following note)</td>
<td>Yes</td>
</tr>
<tr>
<td>ff.pos</td>
<td>Yes</td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>ffb.ksp</td>
<td>Yes</td>
<td>No records</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:** seek operations are supported only to allow for rewind (`seek(fd,0,0)`) and seek-to-end (`seek(fd,0,2)`).

### A.6 The Event Layer

The event layer enables you to monitor, on a per-file basis, the I/O activity that occurs in I/O layer immediately preceding it. It generates statistics as an ASCII log file and reports information such as the number of times an event was called, the event wait time, the number of bytes requested, and so on. You can request the following types of statistics:

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

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

```
setenv FF_IO_LOGFILE +saveIO
```

This layer report counts for `read`, `reada`, `write`, and `writea`. These counts represent the number of calls made to an FFIO layer entry point. In some cases, the system layer may actually use a different I/O system call or use multiple system calls.
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,cachea,event,system

In the above example, the log file will show the I/O activity in the cachea layer.

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nostat</td>
<td>No statistical information is reported.</td>
</tr>
<tr>
<td>summary</td>
<td>Event types that occur at least once are reported.</td>
</tr>
<tr>
<td>brief</td>
<td>A one-line summary for layer activities is reported.</td>
</tr>
</tbody>
</table>

### A.7 The $f77$ Layer

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

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

The keyword syntax is as follows:

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

The type argument specifies the record type and can take one of 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 two 4-byte fields; one at the beginning of the record and one at the end of the 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.

### Table 12. Data Manipulation: *f77* Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
<th>Implementation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Record</td>
<td>Yes</td>
<td>x records</td>
</tr>
</tbody>
</table>

### Table 13. Supported Operations: *f77* Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>fread</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>freada</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
</tr>
<tr>
<td>freadc</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>fwrite</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>fwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
</tr>
<tr>
<td>fwritec</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>fclose</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Operation</td>
<td>Supported</td>
<td>Comments</td>
<td>Required of next lower level?</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>ffweof</td>
<td>Passed through</td>
<td></td>
<td>Yes Only if explicitly requested</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>ffseek(fd,0,0) equals rewind; ffseek(fd,0,2) seeks to end</td>
<td>Yes</td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>Yes</td>
<td>Only in lower-level layer</td>
<td>No</td>
</tr>
</tbody>
</table>

A.8 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.

When used as the last layer above the system or syscall layer, the fd layer supports the assign -B option to enable or disable direct I/O.

All other properties of this layer are the same as the system layer. See Section A.14, page 63 for details.

A.9 The global Layer

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

The syntax for this layer is as follows:

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

The keyword syntax is as follows:

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

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

The `num1` argument specifies the size in 4096-byte blocks of each cache page. `num2` specifies the number of cache pages to be used on each process. If there are `n` processes, then `n × num2` cache pages are used.

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

Table 14. Data Manipulation: `global` Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Stream</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 15. Supported Operations: global Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported operations</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>Always synchronous</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
</tr>
<tr>
<td>ffwritec</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffweof</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>Yes Requires underlying interface to be a stream</td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td>NA</td>
</tr>
</tbody>
</table>

A.10 The ibm Layer

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 syntax is as follows:

```
ibm[.type][.recsize=num1][.mbs=num2]
```
The supported type values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>IBM undefined record type</td>
</tr>
<tr>
<td>f</td>
<td>IBM fixed-length records</td>
</tr>
<tr>
<td>fb</td>
<td>IBM fixed-length blocked records</td>
</tr>
<tr>
<td>v</td>
<td>IBM variable-length records</td>
</tr>
<tr>
<td>vb</td>
<td>IBM variable-length blocked records</td>
</tr>
<tr>
<td>vbs</td>
<td>IBM variable-length blocked spanned records</td>
</tr>
</tbody>
</table>

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 16. Values for Maximum Record Size on \textit{ibm} Layer

<table>
<thead>
<tr>
<th>Field</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)</td>
<td>1</td>
<td>32,760</td>
<td>32,760</td>
<td></td>
</tr>
<tr>
<td>(f)</td>
<td>1</td>
<td>32,760</td>
<td>None</td>
<td>Required</td>
</tr>
<tr>
<td>(fb)</td>
<td>1</td>
<td>32,760</td>
<td>None</td>
<td>Required</td>
</tr>
<tr>
<td>(v)</td>
<td>5</td>
<td>32,756</td>
<td>32,752</td>
<td>Default is \textit{num2}—8 if not specified</td>
</tr>
<tr>
<td>(vb)</td>
<td>5</td>
<td>32,756</td>
<td>32,752</td>
<td>Default is \textit{num2}—8 if not specified</td>
</tr>
<tr>
<td>(vbs)</td>
<td>1</td>
<td>None</td>
<td>None</td>
<td>No maximum record size</td>
</tr>
</tbody>
</table>

### Table 17. Values for Maximum Block Size in \textit{ibm} Layer

<table>
<thead>
<tr>
<th>Field</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)</td>
<td>1</td>
<td>32,760</td>
<td>32,760</td>
<td>Should be equal to \textit{num1}</td>
</tr>
<tr>
<td>(f)</td>
<td>1</td>
<td>32,760</td>
<td>\textit{num1}</td>
<td>Must be equal to \textit{num1}</td>
</tr>
<tr>
<td>(fb)</td>
<td>1</td>
<td>32,760</td>
<td>\textit{num1}</td>
<td>Must be multiple of \textit{num1}</td>
</tr>
<tr>
<td>(v)</td>
<td>9</td>
<td>32,760</td>
<td>32,760</td>
<td>Must be (\geq \textit{num1} + 8)</td>
</tr>
<tr>
<td>(vb)</td>
<td>9</td>
<td>32,760</td>
<td>32,760</td>
<td>Must be (\geq \textit{num1} + 8)</td>
</tr>
<tr>
<td>(vbs)</td>
<td>9</td>
<td>32,760</td>
<td>32,760</td>
<td></td>
</tr>
</tbody>
</table>

### Table 18. Data Manipulation: \textit{ibm} Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
<th>Implementation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Record</td>
<td>No for (f) and (fb) records. Yes for (v), (vb), and (vbs) records.</td>
<td>(f) records for (f) and (fb). (v) records for (u), (v), (vb), and (vbs).</td>
</tr>
</tbody>
</table>
### Table 19. Supported Operations: ibm Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
<th>Required of next lower level?</th>
<th>Used</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffwritec</td>
<td>Yes</td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffweof</td>
<td>Passed through</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>seek(fd, 0, 0) only (equals rewind)</td>
<td>Yes</td>
<td>seek(fd, 0, 0) only</td>
<td></td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td></td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td></td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.11 The mr Layer

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.

**Note:** It is generally more advantageous to configure the layer preceding the mr layer to make the file buffer-resident, assuming that layer can support buffers of sufficient size.
The options are as follows:

\[
\text{mr[.type[.subtype]]:num1:num2:num3}
\]

The keyword syntax is as follows:

\[
\text{mr[.type[.subtype]][.start\_size=num1][.max\_size=num2][.inc\_size=num3]}
\]

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>save</strong></td>
<td>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 <strong>save</strong> option also modifies the behavior of overflow processing. <strong>save</strong> is the default.</td>
</tr>
<tr>
<td><strong>scr</strong></td>
<td>Does not try to load at open and discards data on close (scratch file). The <strong>scr</strong> option also modifies the behavior of overflow processing.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ovfl</strong></td>
<td>Excess data that does not fit in the specified medium is written to the next lower layer. <strong>ovfl</strong> is the default value.</td>
</tr>
<tr>
<td><strong>novfl</strong></td>
<td>When the memory limit is reached, any further operations that try to increase the size of the file fail.</td>
</tr>
</tbody>
</table>

The **num1**, **num2**, and **num3** fields are nonnegative integer values that state the number of 4096-byte blocks to use in the following circumstances:

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>num1</strong></td>
<td>When the file is opened, this number of blocks is allocated for the file. Default: 0.</td>
</tr>
<tr>
<td><strong>num2</strong></td>
<td>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</td>
</tr>
</tbody>
</table>
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: 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>
<thead>
<tr>
<th>Table 20. Data Manipulation: $mr$ Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary function</strong></td>
</tr>
<tr>
<td>Avoid I/O to the extent possible, by holding the file in memory.</td>
</tr>
</tbody>
</table>
Table 21. Supported Operations: mr Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported operations</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fwrite</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fwritea</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fwritec</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fclose</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fflush</td>
<td>Yes</td>
<td>No-op</td>
</tr>
<tr>
<td>ffweof</td>
<td>No</td>
<td>No representation</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>Full support</td>
</tr>
<tr>
<td>(absolute, relative, and from end)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td>No records</td>
</tr>
</tbody>
</table>

A.12 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 line is from a shell script with a file using the assign command and overlying blocking is expected (as specified by BLKTYPO):

```bash
assign -F $BLKTYPO,cos fort.1
```

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

```bash
assign -F null,cos fort.1
```
This is identical to the following command:

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

When used as the last layer above the system or syscall layer, the null layer supports the `assign -B` option to enable or disable direct I/O.

### A.13 The syscall Layer

The syscall layer directly maps each request to an appropriate system call. The layer does not accept any options.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits (1 byte)</td>
<td>Stream (UNICOS/mp system calls)</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td>open</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td>read</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>reada</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td>read plus code</td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td>write</td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td>writea</td>
</tr>
<tr>
<td>ffwritec</td>
<td>Yes</td>
<td>write plus code</td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td>close</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>ffweof</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td>trunc(2)</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>lseek(2)</td>
</tr>
</tbody>
</table>
### Operation Supported Comments

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Lower-level layers are not allowed.

#### A.14 The **system** Layer

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

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

#### A.15 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 syntax is as follows:

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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>Newline-separated records.</td>
</tr>
<tr>
<td>eof</td>
<td>Newline-separated records with a special string such as −e. More than one EOF in a file is allowed.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Record.</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 24. Data Manipulation: text Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported operations</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>Always synchronous</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>fwrite</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
</tr>
<tr>
<td>fwritec</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffweof</td>
<td>Passed through</td>
<td>Yes</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffpos</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 25. Supported Operations: text Layer
A.16 The user and site Layers

The user and site layers let users and site administrators build user-defined or site-specific layers to meet special 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 Appendix B, page 69 for an example of how to create a user FFIO layer.

A.17 The vms Layer

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

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

The following is the alternate keyword syntax for this layer:

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

The following type values are supported:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f</code></td>
<td>VAX/VMS fixed-length records</td>
</tr>
<tr>
<td><code>v</code></td>
<td>VAX/VMS variable-length records</td>
</tr>
<tr>
<td><code>s</code></td>
<td>VAX/VMS variable-length segmented records</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bb</code></td>
<td>Format used for binary blocked transfers</td>
</tr>
</tbody>
</table>
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 $num_1$ field is the maximum record size that may be read or written. It is ignored by the $s$ record type.

Table 26. Values for Record Size: vms Layer

<table>
<thead>
<tr>
<th>Field</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.bb</td>
<td>1</td>
<td>32,767</td>
<td>32,767</td>
<td></td>
</tr>
<tr>
<td>v.tape</td>
<td>1</td>
<td>9995</td>
<td>2043</td>
<td></td>
</tr>
<tr>
<td>v.tr</td>
<td>1</td>
<td>32,767</td>
<td>2044</td>
<td></td>
</tr>
<tr>
<td>s.bb</td>
<td>1</td>
<td>None</td>
<td>None</td>
<td>No maximum record size</td>
</tr>
<tr>
<td>s.tape</td>
<td>1</td>
<td>None</td>
<td>None</td>
<td>No maximum record size</td>
</tr>
<tr>
<td>s.tr</td>
<td>1</td>
<td>None</td>
<td>None</td>
<td>No maximum record size</td>
</tr>
</tbody>
</table>

The $num_2$ field is the maximum segment or block size that is allowed on input and is produced on output. For $vms.f.tr$ and $vms.f.bb$, $num_2$ should be equal to the record size ($num_1$). Because $vms.f.tape$ places one or more records in each block, $vms.f.tape$ $num_2$ must be greater than or equal to $num_1$.

Table 27. Values for Maximum Block Size: vms Layer

<table>
<thead>
<tr>
<th>Field</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.bb</td>
<td>1</td>
<td>32,767</td>
<td>32,767</td>
<td></td>
</tr>
<tr>
<td>v.tape</td>
<td>6</td>
<td>32,767</td>
<td>2,048</td>
<td></td>
</tr>
<tr>
<td>v.tr</td>
<td>3</td>
<td>32,767</td>
<td>32,767</td>
<td>N/A</td>
</tr>
<tr>
<td>s.bb</td>
<td>5</td>
<td>32,767</td>
<td>2,046</td>
<td></td>
</tr>
<tr>
<td>s.tape</td>
<td>7</td>
<td>32,767</td>
<td>2,048</td>
<td></td>
</tr>
<tr>
<td>s.tr</td>
<td>5</td>
<td>32,767</td>
<td>2,046</td>
<td>N/A</td>
</tr>
</tbody>
</table>

For $vms.v.bb$ and $vms.v.disk$ records, $num_2$ 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, \textit{num} 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, \textit{num} is a limit on the block size that will be produced. No limit on record size exists. For \textit{vms.s.tr} and \textit{vms.s.bb}, the block size is an upper limit on the size of a segment. For \textit{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 28. Data Manipulation: \textit{vms} Layer

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Data model</th>
<th>Truncate on write</th>
<th>Implementation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>Record</td>
<td>No for f records. Yes for v and s records.</td>
<td>f records for f formats. v records for v formats.</td>
</tr>
</tbody>
</table>

Table 29. Supported Operations: \textit{vms} Layer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Supported</th>
<th>Comments</th>
<th>Required of next lower level?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffopen</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ffread</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffreada</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
</tr>
<tr>
<td>ffreadc</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ffwrite</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffwritea</td>
<td>Yes</td>
<td>Always synchronous</td>
<td>No</td>
</tr>
<tr>
<td>ffwritec</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ffclose</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffflush</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ffweof</td>
<td>Yes and passed through</td>
<td>Yes for s records; passed through for others</td>
<td>Only if explicitly requested</td>
</tr>
<tr>
<td>ffweod</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ffseek</td>
<td>Yes</td>
<td>seek(fd,0,0) only (equals rewind)</td>
<td>Yes</td>
</tr>
<tr>
<td>Operation</td>
<td>Supported</td>
<td>Comments</td>
<td>Required of next lower level?</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>ffp0s</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ffbksp</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Creating a user Layer  [B]

This appendix explains some of the internals of the FFIO system and explains the ways in which you can put together a user or site layer.

B.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 ffcntl(3c) call.
- Record marks are boundaries between logical records.
- End-of-file (EOF) marks 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 file formats have level numbers in the partitions. FFIO maps 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.

B.1.1 The Operations Structure

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

Table 30 provides 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 ffwrite entry point because it has no representation for EOF. No optional parameters for calls to the internal entry points exist. All arguments are required.

Table 30 lists the variables for the internal entry points and the variable definitions. An internal entry point must be provided for all of these operations:

Table 30. C Program Entry Points

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>The FFI0 pointer (struct fdinfo *)fd.</td>
</tr>
<tr>
<td>file</td>
<td>A char* file.</td>
</tr>
<tr>
<td>flags</td>
<td>File status flag for open, such as O_RDONLY.</td>
</tr>
<tr>
<td>buf</td>
<td>Bit pointer to the user data.</td>
</tr>
<tr>
<td>nb</td>
<td>Number of bytes.</td>
</tr>
<tr>
<td>ret</td>
<td>The status returned; &gt;=0 is valid, &lt;0 is error.</td>
</tr>
<tr>
<td>stat</td>
<td>A pointer to the status structure.</td>
</tr>
<tr>
<td>fulp</td>
<td>The value FULL or PARTIAL defined in fffio.h for full or partial-record mode.</td>
</tr>
</tbody>
</table>
Creating a user Layer

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;ubc</td>
<td>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.</td>
</tr>
<tr>
<td>pos</td>
<td>A byte position in the file.</td>
</tr>
<tr>
<td>opos</td>
<td>The old position of the file, just like the system call.</td>
</tr>
<tr>
<td>whence</td>
<td>The same as the syscall.</td>
</tr>
<tr>
<td>cmd</td>
<td>The command request to the fffcntl(3c) call.</td>
</tr>
<tr>
<td>arg</td>
<td>A generic pointer to the fffcntl argument.</td>
</tr>
<tr>
<td>mode</td>
<td>Bit pattern denoting file’s access permissions.</td>
</tr>
<tr>
<td>argp</td>
<td>A pointer to the input or output data.</td>
</tr>
<tr>
<td>len</td>
<td>The length of the space available at argp. It is used primarily on output to avoid overwriting the available memory.</td>
</tr>
</tbody>
</table>

### B.1.2 FFIO and the stat Structure

The stat structure contains four fields in the current implementation. They mimic the iows structure of the UNICOS/mp 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 the ffio.h file for this purpose.

The fields in the stat structure are as follows:

<table>
<thead>
<tr>
<th>Status field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat.sw_flag</td>
<td>0 indicates outstanding; 1 indicates I/O complete.</td>
</tr>
<tr>
<td>stat.sw_error</td>
<td>0 indicates no error; otherwise, the error number.</td>
</tr>
<tr>
<td>stat.sw_count</td>
<td>Number of bytes transferred in this request. This number is rounded up to the next integral value if a partial byte is transferred.</td>
</tr>
<tr>
<td>stat.sw_stat</td>
<td>This tells the status of the I/O operation. The FFSTAT(stat) macro accesses this field. The following values are valid: FFBOD: At beginning-of-data (BOD).</td>
</tr>
</tbody>
</table>
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.

PFERR: 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 0-byte transfer count.

### B.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 that will fulfill the required functions described in Section B.1.1, page 70 and Section B.1.2, page 71. 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 weak external files in the library. The following script fragment is suggested for UNICOS/mp systems:

```bash
# -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.

ld -o user.o myprog.o
assign -F user,others... fort.1
./abs
```
static char USMID[] = "(#)code/usrbksp.c 1.0 ";
/* COPYRIGHT CRAY INC.*/
/* UNPUBLISHED -- ALL RIGHTS RESERVED UNDER*/
/* THE COPYRIGHT LAWS OF THE UNITED STATES.*/
*/
#include <ffio.h>
#include "usrio.h"
/*
 * trace backspace requests
 */
int
_usrbksp(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 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 INC.
* UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
* THE COPYRIGHT LAWS OF THE UNITED STATES.
*/
#include <ffio.h>
#include "usrrio.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 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)


```c
{
    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(_FPOPEN_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 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 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",
  "fwrite",
  "fwritea",
  "fwritec",
  "fclose",
  "fflush",
  "ffweof",
  "ffweod",
  "ffseek",
  "ffbksp",
  "ffpos",
  "fflistio",
  "ffcntl",
};

/*
* 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;
    sprintf(buf, "TRCX: ret=%ld, stat=%d, err=%d\n",
            ret, stat->sw_stat, stat->sw_error);
    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;
    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_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;
    sprintf(buf, "TRCX: ret=%ld, stat=%d, err=%d\n",
            ret, stat->sw_stat, stat->sw_error);
    write(usr_info->usrfd, buf, strlen(buf));
    #endif
}

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;
    sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
            fio, BPTR2CP(bufptr), nbytes, stat);
    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;
  sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
           fio, BPTR2CP(bufptr), nbytes, stat);
  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;
  sprintf(buf,"(fd / %lx */, &statw[%lx], ",
           fio, stat);
  write(usr_info->usrfd, buf, strlen(buf));
}
static char USMID[] = "@(#)code/ usrread.c 1.0 ";
/* COPYRIGHT CRAY 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->fiopt;
    _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/mp 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); 
    sprintf(buf,"pos %ld, whence %d\n", pos, whence); 
    _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 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
_usrc_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
_usrc_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/mp EOF, or the truncate(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/mp
 * EOF. It is usually mapped to the truncate(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/usrloc.h 1.1 */

/* COPYRIGHT CRAY 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);
This chapter contains information about data conversion, moving data between machines, and implicit and explicit data conversion. It also explains the support provided for reading and writing files in foreign formats, including record blocking and numeric and character conversion.

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

C.1 Conversion Overview

Data can be transferred between UNICOS/mp systems and other computer systems in several ways. These methods include the use of utilities built on TCP/IP (such as ftp). You can also use the data conversion library routines to convert data.

Cray X1 systems support the Institute of Electrical and Electronics Engineers (IEEE) format by default and also support conversion to and from IBM, VAX/VMS, and other formats. For each foreign file type, several supported file and record formats exist or explicit or implicit data conversion can also be used.

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

Explicit conversion is the process by which the user performs calls to subroutines that convert the native data to and from the foreign data formats. These routines are provided for many data formats. This is discussed in more detail in Section C.3.1, page 99.

Implicit conversion is the process by which you 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 data format conversion requires changes in command scripts. This is discussed in more detail in Section C.3.2, page 99.
C.2 Transferring Data

This section describes several ways to transfer data, including using the `fdcp` and other TCP/IP tools.

C.2.1 Using `fdcp` to Transfer Files

The `fdcp(1)` command can handle data that is not a simple disk-resident byte stream. The `fdcp` command assumes that both the data and any record, including an end-of-file (EOF) record, 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. If the file being accessed is on disk and if no `assign -F` attribute is used, the `syscall` layer is used.

C.2.2 Using `ftp` to Move Data between Systems

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.

C.3 Data Item Conversion

The UNICOS/mp operating system provides both the implicit and explicit conversion of data items. Explicit conversion means that your code invokes the routines that convert between native systems and foreign representations.

Options to the `assign(1)` command control 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 the best one for your situation.
C.3.1 Explicit Data Item Conversion

The Cray Fortran library contains a set of subroutines that convert between Cray data formats and the formats of various vendors. These routines are callable from any programming language supported by Cray. The explicit conversion routines convert between IBM, VAX/VMS, or generic IEEE binary data formats and Cray 32-bit IEEE 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.

Table 31 lists the explicit data conversion subroutines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cray X1 systems (IEEE)</th>
<th>Cray -&gt; Foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>IBM2IEG</td>
<td>IEG2IBM</td>
</tr>
<tr>
<td>VAX/VMS</td>
<td>VAX2IEG</td>
<td>IEG2VAX</td>
</tr>
<tr>
<td>IEEE little-endian</td>
<td>IEU2IEG</td>
<td>IEG2IEU</td>
</tr>
<tr>
<td>Cray T3E IEEE (64-bit)</td>
<td>CRI2IEG</td>
<td>IEG2CRI</td>
</tr>
<tr>
<td>SGI MIPS</td>
<td>MIPS2IEG</td>
<td>IEG2MIPS</td>
</tr>
<tr>
<td>User conversion</td>
<td>USR2IEG</td>
<td>IEG2USR</td>
</tr>
<tr>
<td>Site conversion</td>
<td>STE2IEG</td>
<td>IEG2STE</td>
</tr>
</tbody>
</table>

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

C.3.2 Implicit Data Item Conversion

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

The implicit data conversion process is performed in two steps:

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

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

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

If `-N` is used and `-C` is not, an appropriate character conversion type is selected by default, as shown in Table 32.

<table>
<thead>
<tr>
<th><code>-N</code> option</th>
<th><code>-C</code> default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>No numeric conversion</td>
</tr>
<tr>
<td>default</td>
<td>default</td>
<td>No numeric conversion; IEEE 32-bit</td>
</tr>
<tr>
<td>cray</td>
<td>ASCII</td>
<td>Cray “classic” floating-point</td>
</tr>
<tr>
<td>ibm</td>
<td>EBCDIC</td>
<td>IBM 360/370-style numeric conversion</td>
</tr>
<tr>
<td>vms</td>
<td>ASCII</td>
<td>VAX/VMS numeric conversion</td>
</tr>
<tr>
<td>ieee</td>
<td>ASCII</td>
<td>Generic 32-bit IEEE numeric conversion (native)</td>
</tr>
<tr>
<td>ieee_32</td>
<td>ASCII</td>
<td>Alias for <code>ieee</code></td>
</tr>
<tr>
<td>mips</td>
<td>ASCII</td>
<td>SGI MIPS IEEE numeric conversion (128-bit floating-point is “double double” format)</td>
</tr>
<tr>
<td>ieee_64</td>
<td>ASCII</td>
<td>Cray 64-bit IEEE numeric conversion</td>
</tr>
<tr>
<td>ieee_1e</td>
<td>ASCII</td>
<td>Little endian 32-bit IEEE numeric conversion</td>
</tr>
<tr>
<td>ultrix</td>
<td>ASCII</td>
<td>Alias for above</td>
</tr>
<tr>
<td>t3e</td>
<td>ASCII</td>
<td>Cray 64-bit IEEE numeric conversion; denormalized numbers flushed to zero</td>
</tr>
</tbody>
</table>
Cray supports conversion of the supported formats and data types through standard Fortran formatted, unformatted list-directed, and namelist I/O and through `BUFFER IN` and `BUFFER OUT` statements.

Generally, read, write, and rewind are supported for all record formats. Other capabilities such as backspace are usually not available but 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 native representation and a foreign representation, according to its Fortran data type.

If the value in a native element is too large to fit in the foreign element, the foreign element is set to the largest or smallest possible value; no error is generated. When converting from a native element to a smaller foreign element, precision is also lost due to truncation of the floating-point mantissa.

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

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

Files can be converted to either:

- A disk file
- A file transferred from a computer other than the Cray X1 system

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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>t3d</td>
<td>ASCII</td>
<td>Alias for t3e</td>
</tr>
<tr>
<td>user</td>
<td>ASCII</td>
<td>User defined numeric conversion</td>
</tr>
<tr>
<td>site</td>
<td>ASCII</td>
<td>Site defined numeric conversion</td>
</tr>
</tbody>
</table>
For example, if the first read of a foreign file format is like the following example, the library interprets any blocking structures in the file that precede the first data record:

```
INTEGER(KIND=8) INT
REAL(KIND=8) FLOAT1, FLOAT2
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 if the program was compiled with the -sdefault64 compiler option and the INT, FLOAT1, and FLOAT2 variables are default types.

### C.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.

#### C.3.3.1 Explicit Conversion

Explicit data conversion has some distinct advantages, including:

- Providing direct control (including some options not available through implicit conversion) over data conversion
- Allowing programmers to control and schedule the conversion for a convenient and appropriate time
- Performing conversion on large data areas as vector operations, usually increasing performance

One disadvantage of using explicit conversion is that explicit routines require changes to the source code.
C.3.3.2 Implicit Conversion

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

Disadvantages of using implicit conversion include:

- Requiring script changes to the `assign(l)` command
- Making conversion less efficient on a record-by-record basis
- Doing conversion at I/O time according to the declared data types, allowing
  little flexibility for nonstandard requirements

C.3.4 Disabling Conversion Types

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`.

C.4 Foreign Conversion Techniques

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

<table>
<thead>
<tr>
<th>Conversion type</th>
<th>Convert data to/from</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNICOS files</td>
<td>Older Cray UNICOS systems</td>
</tr>
<tr>
<td>IBM conversion</td>
<td>IBM machines</td>
</tr>
<tr>
<td>IEEE conversion</td>
<td>Various types of workstations and different vendors that support IEEE floating-point format</td>
</tr>
<tr>
<td>VAX/VMS conversion</td>
<td>DEC VAX machines that run MVS</td>
</tr>
</tbody>
</table>

C.4.1 UNICOS/mp Conversions

The UNICOS/mp operating system uses `f77` format as the default format for
Fortran unformatted sequential files.

Previous UNICOS operating systems used COS blocking for all blocked files, so
conversion is necessary when moving unformatted, blocked, sequential files from
those Cray systems to the UNICOS/mp operating system. Two common COS
files types require some conversion to make them useful on the UNICOS/mp operating systems.

To read or write unformatted files from UNICOS systems, use one of the following commands:

- If moving a Cray floating point format file from a Cray SV1-series system, use this command:
  
  assign -F cos -N cray cosfile

- If moving an IEEE floating point format file from a Cray SV1-series system, use this command:
  
  assign -F cos -N ieee_64 cosfile

- If moving a file from a Cray T3E system, use this command:
  
  assign -F cos -N t3e cosfile

### C.4.2 IBM Overview

To convert and transfer data between Cray X1 systems and an IBM/MVS or VM (360/370 style) system, you must understand the differences between the UNICOS/mp 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/mp 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/mp text format and to ASCII. For example, ftp performs these conversions and does not require any further conversion on UNICOS/mp systems.

Binary data, however, is more complicated. You must first find a way to transfer the file and to preserve the record boundaries. If workstations are available, this is simple. Few problems are caused by transferring the file and preserving record boundaries.

Cray supports the following IBM record formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Undefined record format</td>
</tr>
<tr>
<td>F</td>
<td>Fixed-length records, one record per block</td>
</tr>
<tr>
<td>FB</td>
<td>Fixed-length, blocked records</td>
</tr>
</tbody>
</table>
C.4.3 IEEE Conversion

By default Cray X1 systems use 32-bit IEEE standard floating point, with two's-complement arithmetic and the ASCII character set. This standard is also used by many workstations and personal computers. 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 Fortran 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:

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

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

On Cray X1 systems, data types can be declared as 32 bits in size and can then be read or written directly. This is the most direct and efficient method to read or write data files for IEEE workstations. The user can alter the declarations of the variables used in the Fortran I/O list to declare them as KIND=4 or as REAL*4 (or INTEGER*4).

For example, to read a file on a Cray X1 system that has 32-bit integers and 32-bit floating-point numbers, consider the following code fragments.

Existing program:

```fortran
REAL    RVAL       ! Default size (32-bits)
INTEGER IVAL       ! Default size (32-bits)
...
READ (1) IVAL, RVAL
```
This program will expect both the integer and floating-point data to be the same size (32 bits). However, it can be modified to explicitly declare the variables to be the same size as the expected data.

Modified program (#1):

```fortran
REAL (KIND=4) RVAL ! Explicit 32-bits
INTEGER (KIND=4) IVAL ! Explicit 32-bits
...
READ (1) IVAL, RVAL
```

This program will correctly read the expected data. However, if this type of modification is too extensive, only the variables used in the I/O statement list need be modified.

Modified program (#2):

```fortran
REAL RVAL ! Default size (32-bits)
INTEGER IVAL ! Default size (32-bits)
REAL (KIND=8) RTMP ! Explicit 64-bits
INTEGER (KIND=4) ITMP ! Explicit 32-bits
...
READ (1) ITMP, RTMP !
```

Change explicitly sized data to default sized data:

```fortran
RVAL = RTMP
IVAL = ITMP
```

On some systems, data types can be declared as 32 bits in size and can then be read or written directly. This is the most direct and efficient method to read or write data files for Cray X1 systems. The user can alter the declarations of the variables used in the Fortran I/O list to declare them as KIND=4 or as REAL*4 (or INTEGER*4).

Other IEEE data conversion variants are also available, but not all variants are available on all systems:

- `ieee` or `ieee_32`: The default workstation conversion specification. Data sizes are based on 32-bit words.
- `ieee_64`: The default IEEE specification on Cray T90/IEEE and Cray T3E systems. Data sizes are based on 64-bit words.
- `ieee_le` or `ultrix`: Data sizes are based on 32-bit words and are little-endian.
mips

Data sizes are based on 32-bit words, except for 128-bit floating point data which uses a “double double” format.

ia

IEEE data types with Intel-style little-endian.

C.4.4 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:

<table>
<thead>
<tr>
<th>Record type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Fixed-length records</td>
</tr>
<tr>
<td>v</td>
<td>Variable-length records</td>
</tr>
<tr>
<td>s</td>
<td>Segmented records</td>
</tr>
<tr>
<td>Media</td>
<td>Definition</td>
</tr>
<tr>
<td>tr</td>
<td>For transparent access to files</td>
</tr>
<tr>
<td>bb</td>
<td>For unlabeled tapes and bb station transfers</td>
</tr>
<tr>
<td>tape</td>
<td>For labeled tapes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>vms.s.tr</td>
<td>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.</td>
</tr>
<tr>
<td>vms.s.tape</td>
<td>Use as an FFIO specification to read or write a file containing segmented records on a labeled tape.</td>
</tr>
<tr>
<td>vms.s.bb</td>
<td>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.</td>
</tr>
</tbody>
</table>

The VAX/VMS system stores its data as a stream of bytes on various devices. Cray X1 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 Cray X1 system makes this byte-ordering transparent when you use text files. When data conversion is used, byte swapping sometimes must be done.
Named pipes, or UNIX FIFO special files for I/O requests, are created with the `mknod` 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 named pipe that can be read to or written from. 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.

Fortran programs can communicate with each other using named pipes. After a named pipe is created, Fortran programs can access that pipe almost as if it were a normal file. The unique aspects of process communication using named pipes are 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 syntax for the command creates a named pipe called `fort.13`. The `p` argument makes it a pipe.

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

A named pipe can be created from within a Fortran program by using the `pxfsystem` function. The following example creates a named pipe:

```
INTEGER ILEN,IERROR
ILEN=0
CALL PXFSYSTEM ('/etc/mknod fort.13 p',ILEN,IERROR)
```

• Fortran programs can use two named pipes: one to read and one to write. A Fortran program can 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. 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 because of 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. An undefined file structure (specified by `assign -s u`) should be specified for a pipe by the sending process. An unblocked structure (specified by `assign -s unblocked`) should be specified for a pipe by the receiving process.

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

  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 an end-of-file (EOF). To detect EOF on a named pipe, the pipe must be opened as read-only by the receiving process. The remainder of this appendix presents more information about detecting EOF.

### D.1 Piped I/O Example without End-of-file Detection

In this example, two Fortran programs communicate without end-of-file (EOF) detection. Program `writerd` generates an array, which contains the elements 1 to 3, and writes the array to named pipe `pipe1`. Program `readwt` reads the three elements from named pipe `pipe1`, prints out the values, adds 1 to each value, and writes the new elements to named pipe `pipe2`. Program `writerd` reads the new values from named pipe `pipe2` and prints them. The `-a` option of the `assign(1)` command allows the two processes to access the same file with different `assign` characteristics.
Example 4: No EOF Detection: program `writerd`

```fortran
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 5: No EOF Detection: program `readwt`

```fortran
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 command sequence executes the programs:

```
ftn -o readwt readwt.f
ftn -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
```
This is the output of the two programs:

\[
\begin{align*}
&\text{ia(1) is 1 in readwt} \\
&\text{ia(2) is 2 in readwt} \\
&\text{ia(3) is 3 in readwt} \\
&\text{ia(1) is 2 in writerd} \\
&\text{ia(2) is 3 in writerd} \\
&\text{ia(3) is 4 in writerd}
\end{align*}
\]

D.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. Read and write 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.

D.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.

This command sequence executes these programs:

```
ftn -o readwt readwt.f
ftn -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 6: EOF Detection: program 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 7: EOF Detection: program readwt

program readwt
parameter(n=3)
dimension ia(n)
C open the pipe as read-only
open(15,form=’unformatted’, action=’read’) ia
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
This 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
End of file detected
application
One or more computer programs that perform specific tasks for a user; these programs use the services of and are under the control of an operating system.

argument
1. A value that determines the result or behavior of a software entity, such as a program or function. 2. A character string in a command line that supplies information for the preceding option. Sometimes called option-argument.

array
A data structure that contains a series of related data items arranged in rows and columns for convenient access. The C shell and the awk(1) command can store and process arrays. A Fortran array is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1 and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

assign environment
The set of information used in Fortran to alter the details of a Fortran connection. This information includes a list of unit numbers, file names, and 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.

asynchronous I/O
I/O operation during which the program performs other operations that do not involve the data in the I/O operation. 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. An operation is complete when all data has been moved.
**batch job**
1. A file of commands that will be executed in batch mode. 2. A group of commands that produces a specific piece of work that the user requires to be executed in a single execution sequence.

**binary blocked**
A file format that describes blocked, nontranslatable data.

**binary stream**
An ordered sequence of characters that can transparently record internal data. Data read in from a binary stream equals data that was written earlier out to that stream under the same implementation.

**block**
1. The smallest allocation unit in a file system; a group of contiguous characters (data bytes). Blocks are separated by some indicator. A block and a physical record are synonymous on magnetic tape. Usually, a block is the size of one physical disk sector. 2. A structure defined by each language processor that represents a contiguous area of memory. Blocks can be local to the defining object module (local blocks) or shared between modules (common blocks). A block can contain instructions, data, or both.

**block size**
The number of words in a defined block.

**blocked**
The state of the delivery of a signal as pending. The state will remain pending until the signal is unblocked.

**blocking**
An optimization that involves changing the iteration order of loops that access large arrays so that groups of array elements are processed as many times as possible while they reside in cache.

**buffer**
In software, a block of memory that the system uses to store data temporarily before it is transmitted to another device or location.
cache
In a processing unit, a high-speed buffer that is continually updated to contain recently accessed contents of memory. Its purpose is to reduce access time.

Caution
In software publications, a hazard statement that highlights information that readers must know to avoid serious, but recoverable errors. (In hardware and training publications, a hazard statement that indicates a potentially hazardous situation that, if not avoided, can result in system damage or data corruption or both.)

coop-array
A syntactic extension to Fortran that offers a method for programming data passing; a data object that is identically allocated on each image and can be directly referenced syntactically by any other image.

contention
Conflict that arises when two or more requests occur simultaneously for a resource that cannot be shared.

CrayDoc
Cray’s documentation system for accessing and searching Cray books, man pages, and glossary terms in HTML and/or PDF format from a web browser. CrayDoc runs on any operating system based on a UNIX or Linux operating system.

data transfer rate
The time required to transfer data from one device to another. Usually, this rate is expressed as a number with the abbreviation Mbps (million bits per second).

data type
A means to categorize data and determine which operations can be applied to the data to get desired results. The Fortran language provides five intrinsic data types: real, integer, complex, logical, and character, and it lets you define additional types.
declaration
A nonexecutable statement that specifies the attributes of a data object. For example, it may be used to specify the type of a variable or function result or the shape of an array.

defered implementation
The label used to introduce information about a feature that will not be implemented until a later release.

direct access
Characterizing data that is intended to be accessed randomly; that is, in no particular order.

direct-access I/O
I/O operation where the a peripheral device or a channel controls data transfer in and out of the computer. The data transfers directly to or from storage and bypasses the processor.

distributed I/O
A method in which the user specifies the data that all tasks will read or write, and the system determines the distribution of work. Also called shared I/O.

entity
1. In Fortran, a general term used to refer to any Fortran concept; for example, a program unit, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface block, a derived type, an I/O unit, a name list group, and so on. 2. In OSI terminology, a layered protocol machine. An entity in a layer performs the functions of the layer in one computer system, accessing the layer entity below and providing services to the layer entity above at local service access points.

entry point
A location in a program or routine at which execution begins. A routine may have several entry points, each serving a different purpose. Linkage between program modules is performed when the linkage editor binds the external references of one group of modules to the entry points of another module.
**environment**
1. The set of hardware and software products on which an application is being run. 2. A set of values supported by the shell used to pass information between processes (specifically, from parent to child).

**environment variable**
A variable that stores a string of characters for use by your shell and the processes that execute under the shell. Some environment variables are predefined by the shell, and others are defined by an application or user. Shell-level environment variables let you specify the search path that the shell uses to locate executable files, the shell prompt, and many other characteristics of the operation of your shell. Most environment variables are described in the ENVIRONMENT VARIABLES section of the man page for the affected command.

**executable file**
A file in a format that can be executed by a computer.

**explicit data conversion**
The process by which the user performs calls to subroutines that convert native data to and from foreign data formats.

**extent**
A structure that defines a starting block and number of blocks for an element of file data.

**file system**
A collection of files and meta information about the files so that they can be accessed and controlled. A file system is mounted to connect it to the overall file system hierarchy and to make it accessible. The root file system is always mounted; other file systems are mounted as needed.

**flag**
Usually a 1-bit table field that indicates whether the condition is true or false.

**flexible file I/O (FFIO)**
A method of I/O, sometimes called layered I/O, wherein each processing step requests one I/O layer or grouping of layers. 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.

floating point
A method of representing data that contains a decimal point.

flush
The operation in which the microprocessor empties the contents of the buffer; the process in which the cache is invalidated.

Fortran standard

function
In C, any called code; in Fortran, a sequence of instructions that returns a value.

I/O
1. Input/output; the transfer of data between devices. 2. The hardware that stores or moves data to/from a processing system.

ID
Identifier or identification.

implicit data conversion
The process by which you 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.

implicit open
The opening of a file or a unit when the first reference to a unit number is an I/O statement other than OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, or REWIND.
inode
A data structure that stores all information about a file (access permissions, file size, file type, owner ID, group ID, and so on) except its name, which is stored in the directory. Each inode has an identifying inode number, which is unique across the file system that includes the file.

kind
Data representation (for example, single precision, double precision). The kind of a type is referred to as a kind parameter or kind type parameter of the type. The kind type parameter KIND indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types.

layered I/O
See flexible file I/O (FFIO).

library
In software, a collection of routines and functions stored in one or more files with the operating system or programming environment. Programming environment libraries link with programs, either at run time, loading, or compiling, to form a complete executable.

load
To create a binary executable file (an executable) from a binary relocatable object file (the object). This process adds library subprograms to the object and resolves the external references among subprograms. Executable files and the libraries and data they access are loaded into memory during the load step. Links are created among modules that must access each other. The command that performs a load is called a link-edit loader, or simply a loader.

named pipe
A first-in, first-out file that allows communication between two unrelated processes running on the same host.

namelist I/O
I/O that 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.
OpenMP
An industry-standard, portable model for shared memory parallel programming.

optimization
The process of changing a program or the environment in which it runs to improve its performance.

page size
The unit of memory addressable through the Translation Lookaside Buffer (TLB). On a Cray X1 system, the base page size is 65,536 bytes, but larger page sizes (up to 4,294,967,296 bytes) are also available.

pattern matching
Recognizing a common code pattern and replacing it with a call to a functionally equivalent library routine.

pointer
A data item that consists of the address of a desired item.

program unit
A Fortran program must contain one main program and can contain any number of the other types of program units. Program units define data environments and the steps necessary to perform calculations. Each program unit has an END statement to terminate the program unit. Each program unit also has an identifying initial statement, but the identifying statement for a main program is optional. A Fortran program unit is either a main program, external subprogram (subroutine or function), module, or block data.

RAID
A standard term for a disk configuration that uses multiple drive spindles to provide data redundancy and/or error correction. There are seven levels of RAID architecture, 0 through 6, of which RAID-3 and RAID-5 are the most common. RAID-3 and RAID-5 each use one extra disk to store parity information needed to recreate data in the event of a single disk failure. RAID-3 uses a dedicated parity disk and is typically faster for throughput-oriented applications, such as file transfer and other sequential applications. RAID-5 distributes the parity information across all disks in the array and is typically faster for transaction processing and other random access applications.
rerun
The process of restarting a job from the beginning.

routine
A section of a program that performs a particular task. It could be a function or a subroutine in Fortran. In C and C++, it is a function.

SHMEM
A library of optimized functions and subroutines that take advantage of shared memory to move data between the memories of processors. The routines can either be used by themselves or in conjunction with another programming style such as Message Passing Interface.

source code
The code that is input to the command language interpreter (in a shell command language) or C or Fortran compiler.

switch
In networks, a device that filters and forwards packets between LAN (local area network) segments.

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

syntax
A formal description of how to enter a command or a programming language statement.

system cache
A set of buffers in kernel memory used for I/O operations by the operating system. 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, however, it is desirable to bypass the system cache and to perform I/O directly between the user’s memory and the logical device.
**system time**
The amount of time that the operating system spends providing services to an application.

**tape block**
A group of contiguous characters recorded on and read from magnetic tape as a unit.

**type**
A means for categorizing data. Each intrinsic and user-defined data type has four characteristics: a name, a set of values, a set of operators, and a means to represent constant values of the type in a program.

**unblocked file structure**
A file that contains undelimited records. Because it does not contain any record control words, it does not have record boundaries.

**unformatted I/O**
Transfer of binary data 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.

**UNICOS**
The operating system for Cray SV1 series systems.

**UNICOS/mp**
The operating system for Cray X1 systems.

**vector**
An array, or a subset of an array, on which a computer operates. When arithmetic, logical, or memory operations are applied to vectors, it is referred to as vector processing.
assign environment
alternate file name, 9
assign command syntax, 5
.buffer file, 23
basic usage, 3
buffer count, 6
buffer size defaults, 19, 22
buffer size specification, 18
C/C++ interface, 3
changing from within a Fortran program, 8
character set conversion, 6
defined, 3
enable/disable direct I/O, 6
FFIO specification, 6
file specification, 6
file structures list, 12
file type, 6
foreign file format specification, 21
foreign numeric conversion, 7
Fortran file truncation, 21
Fortran I/O, 9
Fortran standard specification, 6
IEEE floating point, 7
library buffering, 6, 19
library calling sequence, 8
library routines, 8
list-directed output, 6
local assign mode, 23
memory resident files, 21
opening a file, 4
selecting file structure, 10
setting the FILENV variable, 3, 23
skip unmatched namelist groups, 7
special handling, 6
specify assign object, 7
standard file connection, 6
suppress compressed width, 7
suppress separator in list-directed output, 7
suppress truncate after write, 7
system cache, 20
temporary file, 6
unbuffered I/O, 20
using assign, 3–4
using FFIO in, 3
assign object
add attributes incrementally, 5
defined, 4
file connections, 4
open processing, 4
pattern matching, 4
remove all attributes, 5
replace attributes, 5
unit, 4
view attributes, 5
bin file structure
defined, 14
padding, 14
binary data streams, 32
Block Control Word, 15
blocked file structure
defined, 15
using BUFFER IN/OUT, 15
using ENDFILE, 15
blocked layer
defined, 29
bufa layer, 34
defined, 29
specification, 43
buffers
bufa layer, 43
cache, 17
cachea layer, 18, 20, 47
count, 6
default sizes, 19, 22
defined, 17
enable direct I/O, 18
library, 19
memory-resident files, 58
named pipes, 109
sizes, 6, 18, 33
system cache, 20
unbuffered I/O, 20
usage, 17
using binary stream layers, 32
write-behind and read-ahead, 17

C

cache
defined, 17
hit rate, 17
system, 20
cache layer, 35
defined, 29
improving I/O performance with, 35
specification, 35, 45
cachea layer, 34
defined, 29
specification, 47
calling sequence, 8
Co-array Fortran, 53
conversion methods, 102
COS data conversion, 103
cos file structure
defined, 15
using BUFFER IN/OUT, 15
using ENDFILE IN/OUT, 15
cos layer
defined, 29
specification, 48
creating a user-defined FFIO layer, 69

D
data item conversion
absolute binary files, 103
explicit conversion, 102

implicit conversion, 103
deblocking, 17
debugging
using the event layer to monitor I/O activity, 50
default buffer sizes, 19, 22
direct I/O enable/disable, 6
distributed I/O, 53

E

ebcdic character conversion, 6
environment variables
FILENV, 3
event layer
defined, 29
log file, 50
specification, 50
examples
named pipes, 109
piped I/O with no EOF detection, 110
unformatted direct mr, 37
unformatted sequential mr, 38
user layer, 72

F

f77 layer
defined, 29
specification, 51
fd layer
defined, 29
specification, 53
FFIO
blocked layer, 29
bufa layer, 29
buffer size considerations, 33
cache layer, 29
cachea layer, 18, 29
cachea library buffer, 20
common formats, 31
converting data files, 31
cos layer, 29
creating a user-defined layer, 69
data granularity, 42
defined, 3, 25
event layer, 29
f77 files, 33
f77 layer, 29
fd layer, 29
Fortran I/O forms, 27
global layer, 29
handling binary data, 32
handling multiple EOFs in text files, 31
I/O status fields, 71
ibm layer, 29
layer options, 30
library buffering, 20
list of supported layers, 29
modifying layer behavior, 29
mr layer, 21, 29
null layer, 29
reading and writing f77 files, 33
reading and writing fixed-length records, 32
reading and writing text files, 31
reading and writing unblocked files, 32
removing blocking, 33
selecting file structure, 10
site layer, 29
specifying layers, 28
supported operations, 42
syscall layer, 29
system layer, 28–29
text files, 31
text layer, 29
unblocked files, 32
usage rules, 30
user layer, 29
using, 27
using sequential layers, 26
using the bufa layer, 34
using the cache layer, 35
using the cachea layer, 34
using the global layer, 35
using the mr layer, 35
using the syscall layer, 34
using with assign, 3, 6
vms layer, 29
FFIO and foreign data
foreign conversion tips, 107
IEEE conversion, 105
FFIO layer reference
bufa layer, 43
cache layer, 45
cachea layer, 47
cos layer, 48
event layer, 50
f77 layer, 51
fd layer, 53
global layer, 53
ibm layer, 55
layer definitions, 41
mr layer, 58
null layer, 61
site layer, 65
syscall layer, 62
system layer, 63
text layer, 63
user layer, 65
vms layer, 65
files
alternate file name, 9
bin file structure, 14
blocked file structure, 15
COS blocked, 11
cos file structure, 15
data conversion, 31
default file structure, 11
enabling/suppressing truncation, 21
F77 blocked, 11
foreign format specification, 21
Fortran access methods, 12
handling multiple EOFs in text files, 31
memory-resident, 21
pattern matching, 4
reading and writing f77 files, 33
reading and writing fixed-length records, 32
reading and writing text files, 31
reading and writing unblocked files, 32
sbin file structure, 14
selecting structure, 10
supported file structures, 12
text, 11
text file structure, 15
tuning connections, 9
u (undefined/unknown) file structure, 14
unblocked, 11
unblocked file structure, 13
foreign file conversion
choosing conversion methods, 102
conversion techniques, 103
COS conversions, 103
IBM, 104
IEEE, 105
implicit data item conversion, 99
VAX/VMS, 107
foreign numeric conversion, 7
Fortran
access methods, 22
alternate file name, 9
buffer size specification, 18
co-arrays, 53
default buffer sizes, 19
file access methods, 12
file truncation, 21
I/O extensions, 1
I/O forms, 27
I/O processing, 1
I/O processing with assign, 9
INQUIRE statement, 10
mapping I/O requests to system calls, 28
program interface to assign, 8
selecting file structure, 11
specify standard used, 6
suppress truncate after write, 7
defined, 29
specification, 53
I
I/O processing
log file, 50
overriding defaults, 27
specifying I/O class, 26
unblocked data transfer, 33
I/O processing steps
specifying I/O class, 26
IBM data conversion, 104
ibm layer
defined, 29
specification, 55
IEEE conversion, 105
implicit data item conversion, 99
supported conversions, 101
incrementing assign attributes, 5
L
layered I/O
bufa layer, 43
cache layer, 45
cachea layer, 47
characteristics, 42
cos layer, 48
data model, 42
defined, 25
event layer, 50
f77 layer, 51
fd layer, 53
global layer, 53
ibm layer, 55
implmentation strategy, 42
mr layer, 58
null layer, 61
site layer, 65
site-specific layers, 65
supported layers, 41
supported operations, 42
syscall layer, 62
G
global I/O, 53
global layer, 35
Index

system layer, 63
text layer, 63
user layer, 65
user-defined layers, 65
VMS layer, 65
library
  buffering, 19
  calling sequence, 8
  return status, 9

M
man pages
  asnc1(3f), 23
  asnfile(3f), 8
  asnrm(3f), 8
  assign(1), 1, 5
  assign(3f), 1, 8
  asunuit(3f), 8
  cp(1), 32
  fdfcp(1), 31
  ffassign(3c), 3
  ffassign(3f), 1
  ffopen(3c), 4
  ffread(3c), 27
  ffwrite(3c), 27
  intro_ffio(1), 1
  open(2), 6
memory-resident files, 21
memory-resident layer, 58
MPI, 35, 53
mr layer, 35
  defined, 29
  example, 35
  specification, 58
  unformatted direct, 37
  unformatted sequential, 38
multiple end-of-file records in text files, 31

N
named pipes
  buffers, 109
  creating, 109
defined, 109
detecting EOF, 112
differences from normal I/O, 109
example, 109
piped I/O example (no EOF detection), 110
restrictions, 109
specifying file structure for binary data, 110
null layer
  defined, 29
  specification, 61

O
OpenMP, 53
optimizing
  I/O performance, 33
  text file I/O, 31
  using the event layer to monitor I/O activity, 50

R
raw I/O, 20
read-ahead
  bufa layer, 43
  cachea layer, 47
definition, 17
Record Control Word, 15
removing assign attributes, 5
removing record blocking, 33
replacing assign attributes, 5

S
sbin file structure
  defined, 14
SHMEM, 35, 53
site layer
  defined, 29
  specification, 65
site-specific FFIO layers, 69
supported implicit data conversions, 101
syscall layer
  defined, 29, 34
  specification, 62
system cache, 20
system calls
  in user-defined FFIO layers, 70
system layer
  defined, 28-29
  implicit usage of, 63
  specification, 63

T
text file structure
  defined, 15
  malformed records in Fortran, 15
  using BACKSPACE, 15
  using BUFFER IN/OUT, 15
text layer
  defined, 29
  specification, 63

U
u file structure
  defined, 14
  using BUFFER IN/OUT, 14
  using ENDFILE, 14
unblocked data transfer
  using I/O layers, 33
unblocked file structure
  defined, 13
  padding, 13
  specifications, 13
  using BACKSPACE, 13

using BUFFER IN/OUT, 13
unbuffered I/O, 20
UNIX FFIO special files, 109
user layer
  creating, 69
  defined, 29
  example, 72
  specification, 65
user-defined FFIO layers
  creating, 69
  I/O status fields, 71
  using system calls, 70

V
VAX/VMS
  explicit data item conversion, 99
  record conversion, 107
  transferring files, 98
view current assign attributes, 5
vms layer
  defined, 29
  example, 30
  specification, 65

W
write-behind
  bufa layer, 43
  cachea layer, 47
  defined, 17