Application Programmer's Library Reference Manual

Volume 1

SR–2165 2.0

Cray Research, Inc.

Copyright © 1994, 1995 Cray Research, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

Portions of this manual are based on material reproduced or adapted from IEEE Std 1003.9–1992, copyright (c) 1992 by the Institute of Electrical and Electronics Engineers, Inc. The IEEE takes no responsibility for and will assume no liability for damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, HSX, MPP Apprentice, SSD, SUPERCLUSTER, SUPERSERVER, UniChem, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, Cray NQS, Cray/REELlibrarian, CRAY S-MP, CRAY SUPERSERVER 6400, CRAY T90, CRAY T3D, CrayTutor , CRAY X-MP, CRAY XMS, CRAY-2, CRInform, CRI/*Turbo* Kiva, CS6400, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, HEXAR, IOS, LibSci, ND Series Network Disk Array , Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Research, Inc.

CDC is a trademark and NOS, NOS/BE, and NOS/VE are products of Control Data Corporation. HP is a trademark of Hewlett-Packard Company. HYPERchannel and NSC are trademarks of Network Systems Corporation. IBM is a trademark of International Business Machines Corporation. NASTRAN is a trademark of the National Aeronautics and Space Administration. PostScript is a trademark of Adobe Systems, Inc. Pyramid is a trademark of Pyramid Technology Corporation. SPARC is a trademark of SPARC International, Inc. Sun is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a registered trademark, and the X device is a trademark, of X/Open Company Ltd.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Cray Research Software Documentation Map

The information on the following pages highlights the major body of documentation available for Cray Research customers. The manuals are organized into categories by product designation:

Product	Description
Programming environments	Cray Research programming environments provide an integrated development environment by combining compilers with development tools and libraries.
UNICOS operating system	The UNICOS operating system provides basic system control and management functions. It provides services, including basic batch and interactive system access for users, operating system interfaces for programmers, networking facilities, and sophisticated resource management for administrators. UNICOS MAX, in conjunction with UNICOS on the host system, provides operating system functionality on CRAY T3D systems.
Networking products	Cray Research networking products provide media connections, network programming access, and batch and interactive access for users across a heterogeneous network of computers.
Operating systems supplementary products	The operating systems supplementary products include a variety of products that are used in conjunction with Cray Research operating systems but are not included on the same media.

To use the map, find the product designation closest to your specific needs or role as a Cray Research system user. Documents listed under other categories also may be of interest to you; documents are listed only once, underneath the product to which they most directly apply. The date on the bottom of the page tells you when the information was last revised.

For more information

In addition to the map, you can use the following publications to find documentation specific to your needs:

- *Software Documentation Ready Reference,* publication SQ–2122, serves as a general index to the Cray Research documentation set. The booklet lists documents and man pages according to topic.
- *Software Overview for Users*, publication SG–2052, provides an overview of Cray Research software, including operating systems, programming environments, libraries, tools, applications, and so on. It directs you to documentation that contains user-level information.
- *User Publications Catalog*, publication CP–0099, briefly describes all Cray Research manuals available to you, including some not shown on the map. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on CRInform.

Ordering

To obtain Cray Research publications, order them by publication number from the Distribution Center:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk +1–612–683–5907
Fax number +1–612–452–0141
Email orderdsk@cray.com

10/95

Programming Environments

Ada

Cray Ada Environment, Volume 1: Reference Manual (SR–3014)

Cray Ada Environment, Volume 2: Programming Guide (SR–3082)

Assemblers

Cray Assembler for MPP (CAM) Reference Manual (SR–2510)†

Cray Assembly Language (CAL) for Cray PVP Systems Ready Reference (SQ–3110)

Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual (SR–3108)†

Fortran Compilers

CF90 Ready Reference (SQ–3900)

CF90 Commands and Directives Reference Manual (SR–3901)†

CF90 Fortran Language Reference Manual, Vol. 1 (SR–3902)†

CF90 Fortran Language Reference Manual, Vol. 2 (SR–3903)†

CF90 Fortran Language Reference Manual, Vol. 3 (SR–3905)†

CF90 SPARC Ready Reference (SQ–3904)

Introducing the CF90 SPARC Programming Environment (IN–2155)†

Cray MPP Fortran Reference Manual (SR–2504)†

CF77 Fortran Language Reference Manual (SR–3772)

Cray Standard C Compiler

Cray Standard C Reference Manual (SR–2074)†

Cray Standard C Ready Reference (SQ–2076)

Cray Standard C Reference Manual for MPP (SR–2506)†

C++ Compiler

Cray C/C++ Reference Manual (SR–2179)†

Cray C/C++ Ready Reference (SQ–2180)

CrayLibs

Application Programmer's I/O Guide (SG–2168)†

Application Programmer's Library Reference Manual (SR–2165)★

Introducing CrayLibs (IN–2167)†

Math Library Reference Manual (SR–2138)★

Parallel Virtual Machine (PVM) Reference Card (SQ–2512)

PVM and HeNCE Programmer's Manual (SR–2501)†

Scientific Libraries Reference Manual (SR–2081)★

CrayTools

Compiler Information File (CIF) Reference Manual (SR–2401)†

Cray TotalView Debugger Reference Card (SQ–2519)

Guide to Parallel Vector Applications (SG–2182)†

Introducing the Cray TotalView Debugger (IN–2502)†

Introducing the MPP Apprentice Tool (IN–2511)†

Introducing the Program Browser (IN–2140)†

UNICOS CDBX Symbolic Debugger Reference Card (SQ–2110)

UNICOS CDBX Symbolic Debugger Reference Manual (SR–2091)†

UNICOS CDBX Debugger User's Guide (SG–2094)†

General

Cray Research MPP Software Guide (SG–2508)†

CRAY T3D Emulator User's Guide (SG–2500)†

Installing Programming Environment Products (SG–5191)

Migrating to the CRAY T90 Series IEEE Floating Point (SN–2194)†

Optimizing Code on Cray PVP Systems (SG–2192)†

Programming Environment Installation Guide (SG–5217)

Message Passing Toolkit

MPT: PVM Programmer's Manual (SR–2196)†

MPT: MPI Programmer's Manual (SR–2197)†

† Available with CrayDoc
★ Man pages available with the `man` command

UNICOS Operating System

General UNICOS

Software Documentation Ready Reference (SQ–2122)

Software Overview for Users (SG–2052)†

User's Guide to Online Information (SG–2143)†

UNICOS `ed` Reference Card (SQ–2055)

UNICOS Environment Variables Ready Reference (SQ–2117)

UNICOS File Formats and Special Files Reference Manual (SR–2014)★

UNICOS Standard Shell Reference Card (SQ–2115)

UNICOS User Commands Ready Reference (SQ–2056)★

UNICOS User Commands Reference Manual (SR–2011)★

UNICOS `vi` Reference Card (SQ–2054)

UNICOS Visual Interfaces User's Guide (SG–3094)†

Networking Feature Use and Administration

Front-end Protocol Internal Reference Manual (SM–0042)

FTA User and Administrator Manual (SG–2144)†

Kerberos Administrator's Guide (SG–2306)†

Kerberos User's Guide (SG–2409)†

Network Queuing System (NQS) User's Guide (SG–2105)†

ONC+ Technology for the UNICOS Operating System (SG–2169)†

Remote Procedure Call (RPC) Reference Manual (SR–2089)†

TCP/IP Network User's Guide (SG–2009)†

UNICOS Networking Facilities Administrator's Guide (SG–2304)†

UNICOS NQS and NQE Administrator's Guide (SG–2305)†

UNICOS Station Call Processor (USCP) Administrator's Guide (SG–2308)†

UNICOS MAX

Cray MPP Loader User's Guide (SG–2514)†

Cray MPP Simulator User's Guide (SG–2503)†

CRAY T3D Administrator's Guide (SG–2507)†★

UNICOS Basic Administration and Operation

CRAY EL Series IOS Commands Ready Reference (SQ–2162)★

CRAY EL Series IOS Commands Reference Manual (SR–2408)★

CRAY EL Series IOS Messages (SQ–2402)

CRAY IOS-V Commands Reference Manual (SR–2170)★

CRAY IOS-V Messages (SQ–2172)†

General UNICOS System Administration (SG–2301)†

`nmake` Reference Card (SQ–2146)

OLNET Online Diagnostic Network Communications Program Maintenance Manual for UNICOS (SD–1021)†

Online Maintenance Tools Guide for Cray PVP Systems (SD–1012)†

UNICOS Basic Administration Guide for CRAY J90 Series and CRAY EL Series (SG–2416)†

UNICOS Configuration Administrator's Guide (SG–2303)†

UNICOS Installation Guide (SG–2112)

UNICOS Installation Guide for the CRAY EL Series (SG–5201)

UNICOS Installation Guide for the CRAY J90 Series (SG–5271)

UNICOS Installation Menu System Reference Card (SQ–2411)

UNICOS Installation/Configuration Menu System User's Guide (SG–2412)†

UNICOS Resource Administration (SG–2302)†

UNICOS Shared File System (SFS) Administrator's Guide (SG–2114)†

UNICOS Feature Use and Administration

UNICOS Administrator Commands Reference Manual (SR–2022)★

UNICOS Administrator Commands Ready Reference (SQ–2413)★

UNICOS Message System Programmer's Guide (SG–2121)†

UNICOS Multilevel Security (MLS) Feature User's Guide (SG–2111)†

UNICOS System Security Overview for Administrators (SG–2141)†

UNICOS Tape Subsystem Administrator's Guide (SG–2307)†

UNICOS Tape Subsystem User's Guide (SG–2051)†

UNICOS under UNICOS Administrator's Guide (SG–2156)†

† Available with CrayDoc
★ Man pages available with the `man` command

UNICOS Operating System (continued)

Programming Support

Segment Loader (SEGLDR) and `ld` Reference Manual (SR–0066)†

SEGLDR Reference Card (SQ–0303)

UNICOS System Libraries Ready Reference (SQ–2147)★

UNICOS System Libraries Reference Manual (SR–2080)★

Programming Support (continued)

UNICOS Macros and Opdefs Reference Manual (SR–2403)★

UNICOS System Calls Reference Manual (SR–2012)★

UNICOS X Window System Reference Manual (SR–2101)†

X Window System Resources Ready Reference (SQ–2123)

Networking Products

ATM

Asynchronous Transfer Mode (ATM) Administrator's Guide (SG–2193)†

fy Driver

fy Driver Administrator's Guide (SG–2132)

HIPPI

HIPPI Channel Administrator's Guide (SG–2159)

Network Queuing Environment (NQE)

NQE User's Guide (SG–2148)†

NQE Ready Reference (SG–2149)

NQE Administration (SG–2150)†

Introducing NQE (IN–2153)†

RQS MVS

RQS User's Guide for IBM MVS Systems (SG–2405)

RQS Administrator's Guide for IBM MVS Systems (SG–2406)

Fiber Distributed Data Interface (FDDI)

Fiber Distributed Data Interface (FDDI) Administrator's Guide (SG–3091)

RQS VMS

RQS/NQC User's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3151)

RQS/NQC Administrator's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3152)

SUPERLINK VM

SUPERLINK VM User's Guide: RQS Component (SI–0170)

SUPERLINK VM Administrator's Guide: RQS Component (SI–0171)

SUPERLINK VMS

SUPERLINK Administrator's Guide for DEC VAX/VMS Systems (SV–3154)

SUPERLINK User's Guide for DEC VAX/VMS Systems (SV–3153)

Operating Systems Supplementary Products

CCMT

Concurrent Maintenance Tools (CCMT) Guide (SD–1027)†

Cray/REELlibrarian (CRL)

Cray/REELlibrarian (CRL) Administrator's Guide (SG–2127)

Cray/REELlibrarian (CRL) User's Guide (SG–2126)

SMARTE

System Maintenance and Remote Testing Environment (SMARTE) Guide (SD–1017)†

Data Migration Facility (DMF)

Cray Data Migration Facility (DMF) Administrator's Guide (SG–2135)†★

Cray Data Migration (DMF) MSP Writer's Guide (SN–2098)†

OWS

Support System Reference Manual (SR–3077)★

Support System Operator's Guide (SG–3078)†

Support System Administrator's Guide (SG–3079)†

Support System Ready Reference (SQ–3080)

† Available with CrayDoc
★ Man pages available with the `man` command

New Features

Application Programmer's Library Reference Manual SR–2165 2.0.

This manual describes the Application Library for CrayLibs 2.0. The changes to the manual are detailed below.

The shared memory routines, which describe data passing, were added for the Programming Environment 1.2.1 release and were released in online-only format at that time. They are printed for the first time in this manual for the CrayLibs 2.0 release. The following is a list of these routines; for information about the routines, see the individual man pages or the INTRO_SHMEM(3) man page:

SHMEM_AND(3)	SHMEM_OR(3)
SHMEM_BARRIER(3)	SHMEM_PROD(3)
SHMEM_BROADCAST(3)	SHMEM_PUT(3)
SHMEM_CACHE(3)	SHMEM_QUIET(3)
SHMEM_COLLECT(3)	SHMEM_SUM(3)
SHMEM_IPUT(3)	SHMEM_SWAP(3)
SHMEM_IXPUT(3)	SHMEM_WAIT(3)
SHMEM_LOCK(3)	SHMEM_XOR(3)
SHMEM_MAX(3)	SHMEM_STACK(3)
SHMEM_MIN(3)	INTRO_SHMEM(3)
START_PES(3)	NUM_PES(3)
SHMEM_MY_PE(3)	

The following new man pages were added for the CrayLibs 2.0 release to document new routines and are printed in this manual:

ffiolock(3C)	FSUP(3F)
IEG2CRI(3F)	IEG2CRI_77(3F)

The CF90 intrinsic procedures are no longer printed in this manual; they are printed in the *CF90 Fortran Language Reference Manual, Volume 2,* publication SR-3903.

The AQOPENM routine was added at this release, and is documented on the AQOPEN man page.

In addition, the CALLSEQ(3F) man page, which describe the calling sequence for subprograms, was rewritten for the CrayLibs 2.0 release.

Record of Revision

The date of printing or software version number is indicated in the footer.

Version	*Description*
1.2	October 1994. Original printing. This manual supports the Programming Environment 1.2 release. Many of the routines in this manual were originally printed in the *UNICOS Fortran Library Reference Manual*, publication SR–2079, and the *UNICOS C Library Reference Manual*, publication SR–2080. Change bars have been added to indicate technical changes on those pages and to indicate new pages. See the New Features page for details.
2.0	November 1995. Reprint to support the Programming Environment 2.0 release. This manual is split into 2 volumes for this release; for information about the content of each volume, see the "Preface" to this manual or the `INTRO_APPLIBS`(3F) man page.

Preface

This publication documents Fortran-callable subprograms and routines available to users of the Cray Research systems for the Programming Environment 2.0 release. It supplements information contained in other manuals of the Programming Environment documentation set.

This is a reference manual for application and system programmers. Readers should also have a working knowledge of either the UNICOS or the UNIX operating system and a working knowledge of the Fortran or C programming languages.

Portions of this manual are based on material reproduced or adapted from IEEE Std 1003.9-1992, copyright (c)1992 by the Institute of Electrical and Electronics Engineers, Inc. The IEEE takes no responsibility for and will assume no liability for damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

The features documented for Programming Environment releases are relevant for Cray PVP and Cray MPP systems only; they are not valid for CRAY X-MP or CRAY-2 systems. See the information about machine naming conventions in this Preface for a complete definition of Cray Research hardware platforms.

Documentation organization

The printed versions of the Application Library routines appear in 2 volumes and are grouped according to topics.

Within each topic section, routines are further grouped into the following categories:

- Common routines which run on all Cray Research systems
- Routines that run only on Cray MPP systems
- Routines that run only on Cray PVP systems

Volume 1 contains the following topic sections:

- Calling Sequence for Cray PVP routines
- I/O routines
- FFIO routines
- SHMEM (Shared Memory) routines
- Synchronization routines for Cray MPP systems
- Special Library routines, including character functions, packing routines, timing routines, and byte and bit manipulation routines

Volume 2 contains the following topic sections:

- Calling Sequence for Cray PVP routines
- Data conversion routines
- Heap, table, and data segment management routines
- Programming aids, including tracing and timing routines, watchpointing routines, traceback routines, dump routines, and signal routines.
- Interface routines, including job control routines, floating-point interrupt routines, and special-purpose interface routines.
- Search and sort routines

Related publications

The following documents contain additional information that may be helpful:

- *Segment Loader (SEGLDR) and `ld` Reference Manual*, publication SR–0066
- *UNICOS User Commands Reference Manual*, publication SR–2011
- *Software Overview for Users*, publication SG–2052
- *UNICOS User Commands Ready Reference*, publication SQ–2056

- *UNICOS Performance Utilities Reference Manual*, publication SR–2040
- *Interlanguage Programming Conventions*, publication SN–3009
- *UNICOS Visual Interfaces User's Guide*, publication SG–3094
- *UNICOS Tape Subsystem User's Guide*, publication SG–2051
- *UNICOS System Libraries Reference Manual*, publication SR–2080
- *UNICOS System Calls Reference Manual*, publication SR–2012
- *Introducing CrayLibs*, publication IN–2167
- *Application Programmer's I/O Guide*, publication SG–2168
- *UNICOS CDBX Symbolic Debugger Reference Manual*, publication SR–2091
- *UNICOS CDBX Debugger User's Guide*, publication SG–2094
- *Introducing the Program Browser*, publication IN–2140
- *Optimizing Code on Cray PVP Systems*, publication SG–2192

In addition to these documents, several documents are available that describe Cray Research's compiler systems. Some of these manuals are as follows:

- *CF90 Ready Reference*, publication SQ–3900
- *CF90 Commands and Directives Reference Manual*, publication SR–3901
- *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902
- *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903
- *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905
- *CF77 Commands and Directives*, publication SR–3771
- *CF77 Fortran Language Reference Manual*, publication SR–3772

Ordering publications

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

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

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

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

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage(`*x*`)`	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files)

Convention	Meaning
	4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands Some internal routines (for example, the `ddcnt()` routine) do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command line.
...	Ellipses indicate that a preceding command-line element can be repeated.

The following machine naming conventions may be used throughout this document:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems)

Term	Definition
	CRAY J90 series (CRAY J916 and CRAY J932 systems)
	CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems)
	CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems)
	CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

The default shell in the UNICOS operating system, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS operating system also supports the optional use of the C shell.

Cray UNICOS Version 9.0 is an X/Open Base 95 branded product.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands* . Both terms may appear in this document.

Man page sections

The entries in this document are based on a common format. The following list shows the order of sections in an entry and describes each section. Most entries contain only a subset of these sections.

Section heading	Description
NAME	Specifies the name of the entry and briefly states its function.
SYNOPSIS	Presents the syntax of the entry.
IMPLEMENT ATION	Identifies the Cray Research systems to which the entry applies.
STANDARDS	Provides information about the portability of a utility or routine.
DESCRIPTION	Discusses the entry in detail.
NOTES	Presents items of particular importance.
CAUTIONS	Describes actions that can destroy data or produce undesired results.
WARNINGS	Describes actions that can harm people, equipment, or system software.
ENVIRONMENT VARIABLES	Describes predefined shell variables that determine some characteristics of the shell or that affect the behavior of some programs, commands, or utilities.
RETURN VALUES	Describes possible return values that indicate a library or system call executed successfully, or identifies the error condition under which it failed.
EXIT STATUS	Describes possible exit status values that indicate whether the command or utility executed successfully .

Section heading	Description
MESSAGES	Describes informational, diagnostic, and error messages that may appear. Self-explanatory messages are not listed.
FORTRAN EXTENSIONS	Describes how to call a system call from Fortran. Applies only to system calls.
BUGS	Indicates known bugs and deficiencies.
EXAMPLES	Shows examples of usage.
FILES	Lists files that are either part of the entry or are related to it.
SEE ALSO	Lists entries and publications that contain related information.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a document online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a document. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

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

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to the Internet, using the following Internet address:

  `publications@timbuk.cray.com`

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

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

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

  +1–612–683–5600

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

- Use the postage-paid Reader's Comment Form at the back of the printed document.

We value your comments and will respond to them promptly.

CONTENTS

intro_applibs.3f ..... Introduction to Cray Research Application Programmer's library 1

Calling Sequence

callseq ..... Details the calling sequence for Cray PVP systems 5

I/O routines

intro_io, INTRO_IO ..... Introduction to I/O routines 11

Common pages

aqclose, AQCLOSE ..... Closes an asynchronous queued I/O file 25
aqopen, AQOPEN, ASOPENM ..... Opens a file for asynchronous queued I/O 27
aqread, AQREAD, AQREADC ..... Queues a simple or compound asynchronous I/O read request 29
aqrecall, AQRECALL ..... Waits for completion of asynchronous queued I/O request 31
aqstat, AQSTAT ..... Checks the status of asynchronous queued I/O requests 33
aqwait, AQWAIT ..... Waits for completion of asynchronous queued I/O requests 35
aqwrite, AQWRITE, AQWRITEC ..... Queues a simple or compound asynchronous I/O write request 37
asnctl, ASNCTL ..... Controls function of ASSIGN, ASNFILE, ASNUNIT, and ASNRM routines 39
asnqfile, ASNQFILE, ASNQUNIT ..... Returns the assign options currently in effect for a file name or unit number 41
assign, ASSIGN, ASNUNIT, ASNFILE, ASNRM ... Provides library interface to assign processing 43
asyncms, ASYNCMS, ASYNCDR ..... Sets I/O mode for random-access routines to asynchronous 45
checkms, CHECKMS, CHECKDR ..... Checks status of asynchronous random-access I/O operation 47
closev, CLOSEV ..... Closes volume and mounts next volume in Volume Identifier list 49
closms, CLOSMS, CLOSDR ..... Writes master index and closes random-access file 50
findms, FINDMS ..... Reads record into data buffers used by random-access routines 52

flush, FLUSH ..... Writes data buffered by Fortran output statements to a file 54
fsup, FSUP, ISUP ..... Suppress values in Fortran edit-directed output 56
gettp, GETTP ..... Gets information about an opened tape file 57
getwa, GETWA, SEEK ..... Synchronously and asynchronously reads data from the word-addressable, random-access file 61
gtstdptr, GTSTDPTR ..... Returns pointer to standard file 64
numblks, NUMBLKS ..... Returns the current size of a file in 512-word blocks 65
openms, OPENMS, OPENDR ..... Opens a local file as a random-access file that can be accessed or changed by the record-addressable, random-access file I/O routines 66
putwa, PUTWA, APUTWA ..... Writes to a word-addressable, random-access file 69
read, READ, READP ..... Reads words, full or partial record modes 71
readc, READC, READCP ..... Reads characters, full or partial record mode 73
readms, READMS, READDR ..... Reads a record from a random-access file to memory 75
stindx, STINDX, STINDR ..... Allows an index to be used as the current index by creating a subindex 78
syncms, SYNCMS, SYNCDR ..... Sets I/O mode for random-access routines to synchronous 81
waitms, WAITMS, WAITDR ..... Waits for completion of an asynchronous I/O operation 83
wclose, WCLOSE ..... Finalizes changes and closes word-addressable, random-access file 85
wopen, WOPEN ..... Opens a word-addressable, random-access file 87
write, WRITE, WRITEP ..... Writes words, full or partial record mode 90
writec, WRITEC, WRITECP ..... Writes characters, full or partial record mode 92
writms, WRITMS, WRITDR ..... Writes to a random-access file on disk 94

PVP Pages

acptbad, ACPTBAD ..... Makes bad data available 103
checktp, CHECKTP ..... Checks tape position 106
endsp, ENDSP ..... Disables special tape processing 107
readibm, READIBM ..... Reads two IBM 32-bit floating-point words from each Cray Research 64-bit word 109
rnl, RNLFLAG, RNLDELM, RNLSEP, RNLREP, RNLCOMM ..... Manipulates characters recognized by NAMELIST 110
rnlecho, RNLECHO ..... Specifies output unit for NAMELIST error messages and echo lines 112
rnlskip, RNLSKIP ..... Takes appropriate action when an undesired NAMELIST group is encountered 113

rnltype, RNLTYPE ..... Determines action if type mismatch occurs across equal sign on NAMELIST input record 114
setsp, SETSP ..... Enables and disables EOV processing 115
settp, SETTP ..... Positions a tape file at a tape block and/or a tape volume 117
skipbad, SKIPBAD ..... Skips bad data 120
skipf, SKIPF ..... Skips files 122
startsp, STARTSP ..... Enables special tape processing 124
tsync, TSYNC ..... Requests tape synchronization 126
wnl, WNLFLAG, WNLDELM, WNLSEP, WNLREP ..... Provides user control of NAMELIST output format 128
wnlline, WNLLINE ..... Allows each NAMELIST variable to begin on a new line 130
wnllong, WNLLONG ..... Selects NAMELIST output line length 131
writibm, WRITIBM ..... Writes two IBM 32-bit floating-point words from each Cray 64-bit word 132

FFIO pages

intro_ffio, INTRO_FFIO ..... Describes performance options available with the FFIO layers 133

Common pages

fffcntl ..... Performs functions on files opened using flexible file I/O 145
ffiolock, ffiounlock ..... Provide locking for FFIO functions 153
fflistio ..... Initiates a list of I/O requests using flexible file I/O 154
ffopen, ffopens, ffclose ..... Opens or closes a file using flexible file I/O 158
ffpos ..... Positions files opened using flexible file I/O 162
ffread, ffwrite, ffweof, ffweod ..... Provides flexible file I/O 168
ffreada ..... Provides asynchronous read using flexible file I/O 171
ffseek, ffbksp ..... Repositions a flexible file I/O file 175
ffwritea ..... Provides asynchronous write using flexible file I/O 179
ffsetsp ..... Initiates EOV processing for files opened using flexible file I/O 181

SHMEM ROUTINES

intro_shmem ..... Introduction to logically shared memory access routines 183

MPP pages

num_pes ..... Returns the number of processing elements (PEs) running in an application 191

shmem_and, shmem_short_and_to_all, shmem_int_and_to_all, shmem_int8_and_to_all, shmem_int4_and_to_all ..... Performs a logical AND function across a set of processing elements (PEs) 192

shmem_barrier ..... Performs a barrier operation on a subset of processing elements (PEs) 197

shmem_broadcast, shmem_broadcast32, shmem_broadcast4 ..... Broadcasts a block of data from one processing element (PE) to one or more target PEs 200

shmem_cache, shmem_set_cache_inv, shmem_set_cache_line_inv, shmem_clear_cache_inv, shmem_udcflush, shmem_udcflush_line ..... Controls data cache utilities 204

shmem_collect, shmem_fcollect, shmem_collect32, shmem_fcollect32, shmem_collect4, shmem_fcollect4 ..... Concatenates blocks of data from multiple processing elements (PEs) to an array in every PE 207

shmem_iget, shmem_iput, shmem_iget32, shmem_iput32, shmem_iget4, shmem_iput4 ..... Transfers strided data between remote and local memory 212

shmem_ixget, shmem_ixput, shmem_ixget32, shmem_ixput32, shmem_ixget4, shmem_ixput4 ..... Transfers strided data between remote and local memory 216

shmem_lock, shmem_clear_lock, shmem_set_lock, shmem_test_lock ..... Releases, locks, and tests a mutual exclusion memory lock 220

shmem_max, shmem_short_max_to_all, shmem_int_max_to_all, shmem_int8_max_to_all, shmem_comp8_max_to_all, shmem_int4_max_to_all, shmem_real4_max_to_all ..... Performs a maximum function reduction across a set of processing elements (PEs) 222

shmem_min, shmem_short_min_to_all,
shmem_int_min_to_all,
shmem_float_min_to_all,
shmem_double_min_to_all,
shmem_int8_min_to_all,
shmem_real8_min_to_all,
shmem_int4_min_to_all,
shmem_real4_min_to_all ..... Performs a minimum function reduction across a set of processing elements (PEs) 228
shmem_my_pe, shmem_n_pes ..... Returns processing element (PE) information 234
shmem_or, shmem_short_or_to_all,
shmem_int_or_to_all,
shmem_int8_or_to_all,
shmem_int4_or_to_all ..... Performs a logical OR function reduction across a set of processing elements (PEs) 236
shmem_prod, shmem_short_prod_to_all,
shmem_int_prod_to_all,
shmem_float_prod_to_all,
shmem_double_prod_to_all,
shmem_complexf_prod_to_all,
shmem_complexd_prod_to_all,
shmem_int8_prod_to_all,
shmem_real8_prod_to_all,
shmem_comp8_prod_to_all,
shmem_int4_prod_to_all,
shmem_real4_prod_to_all,
shmem_comp4_prod_to_all ..... Performs a product reduction across a set of processing elements (PEs) 241
shmem_get, shmem_put, shmem_get32,
shmem_put32, shmem_get4, shmem_put4 ..... Transfers data between local and remote memory .. 247
shmem_quiet ..... Waits for completion of all outstanding remote writes issued by a processing element (PE) 252
shmem_stack ..... Makes a stack address remotely accessible 254

shmem_sum, shmem_short_sum_to_all,
shmem_int_sum_to_all,
shmem_float_sum_to_all,
shmem_double_sum_to_all,
shmem_complexf_sum_to_all,
shmem_complexd_sum_to_all,
shmem_int8_sum_to_all,
shmem_real8_sum_to_all,
shmem_comp8_sum_to_all,
shmem_int4_sum_to_all,
shmem_real4_sum_to_all,
shmem_comp4_sum_to_all ..... Performs a sum reduction across a set of processing elements (PEs) 255
shmem_swap ..... Performs an atomic swap to a remote address 262
shmem_wait, shmem_wait_until ..... Waits for a variable on the local processing element (PE) to change 264
shmem_xor, shmem_short_xor_to_all,
shmem_int_xor_to_all,
shmem_int8_xor_to_all,
shmem_real8_xor_to_all,
shmem_comp8_xor_to_all,
shmem_int4_xor_to_all,
shmem_real4_xor_to_all,
shmem_comp4_xor_to_all ..... Performs a reduction across a set of processing elements (PEs) 268
start_pes ..... Starts a processing element (PE) for a message- or data-passing program 274

Synch routines

intro_sync, INTRO_SYNC ..... Introduction to synchronization routines 277

MPP pages

barrier ..... Registers the arrival of a task at a barrier and suspends task execution until all other tasks arrive at the barrier 281
clear_event ..... Clears the eureka event and returns control to the calling task 283
clear_event, CLEAR_EVENT ..... Clears an event and returns control to the calling task 285
clear_lock, CLEAR_LOCK ..... Clears a lock and returns control to the calling task 288
set_barrier ..... Registers the arrival of a task at a barrier 290
set_barrier, SET_BARRIER ..... Registers the arrival of a task at a barrier 291

set_event, SET_EVENT ..... Posts an event and returns control to the calling task 292
set_lock, SET_LOCK ..... Sets a lock and returns control to the calling task .. 295
test_barrier ..... Tests a barrier to determine its state (set or cleared) 296
test_barrier, TEST_BARRIER ..... Tests a barrier to determine its state (set or cleared) 297
test_event ..... Returns the state of the eureka event, either posted or cleared 298
test_event, TEST_EVENT ..... Returns the state of an event 299
test_lock, TEST_LOCK ..... Sets a lock and returns the lock state 301
wait_barrier ..... Suspends task execution until all tasks arrive at the barrier 302
wait_barrier, WAIT_BARRIER ..... Suspends task execution until all tasks arrive at the barrier 303
wait_event ..... Delays the calling task until the eureka event is posted 304
wait_event, WAIT_EVENT ..... Delays the calling task until the specified event is posted 305

Special library routines

intro_specialibs, INTRO_SPECIALIBS ..... Introduction to specialized libraries and miscellaneous routines 309

Common pages

iceil, ICEIL ..... Returns integer ceiling of a rational number 315
ipxfargc, IPXFARGC ..... Returns number of command-line arguments excluding the command name 316
mov, STRMOV, MOVBIT, MOVBITZ ..... Moves bytes or bits from one variable or array to another 317
pack, PACK ..... Compresses stored data 320
pxfaccess, PXFACESS ..... Checks accessibility of a named file 322
pxfchdir, PXFCHDIR ..... Change current directory to a specified directory 324
pxfchmod, PXFCHMOD ..... Sets file modes for a named file 326
pxfchown, PXFCHOWN ..... Changes owner and group of a file 329
pxfchroot, PXFCHROOT ..... Changes root directory to specified directory 331
pxfconst, PXFCONST ..... Returns value associated with symbolic constants .. 333
pxfcreat, PXFCREAT ..... Creates a new file or rewrites an existing file 337
pxffcntl, PXFFCNTL ..... Provides a subset of fcntl(2) functionality, except the third argument is always an integer 340
pxffileno, PXFFILENO ..... Returns the file descriptor for a specified unit 343
pxfgetarg, PXFGETARG ..... Returns command-line argument 344
pxfgetenv, PXFGETENV ..... Returns value for environment name 346

pxfintget, PXFINTGET ..... Allows values stored in individual components of a structure to be extracted and used 349
pxfintset, PXFINTSET ..... Allows components of a structure to be set or modified 351
pxflink, PXFLINK ..... Creates a link to a file 352
pxfopen, PXFOPEN ..... Provides a Fortran interface to the open(2) system call 354
pxfrename, PXFRENAME ..... Renames a file 357
pxfrmdir, PXFRMDIR ..... Removes a directory entry 359
pxfstat, PXFSTAT ..... Retrieves the file status 361
pxfstrget, PXFSTRGET ..... Allows values stored in individual components of a structure to be extracted and used 363
pxfstrset, PXFSTRSET ..... Allows values stored in individual components of a structure to be set 365
pxfstructcreate, PXFSTRUCTCREATE ..... Creates an instance of the desired structure and returns a nonzero handle in the argument *jhandle* .. 367
pxfstructfree, PXFSTRUCTFREE ..... Deletes the instance of the structure referenced by *jhandle* 369
pxfuname, PXFUNAME ..... Retrieves the operating system name 371
pxfunlink, PXFUNLINK ..... Removes a directory entry 373
secondr, SECONDR ..... Returns elapsed wall-clock time in seconds 375
sysclock, SYSCLOCK ..... Returns real-time clock value and number of wraps 377
timef, TIMEF ..... Returns elapsed wall-clock time since the call to TIMEF 378
trimlen, TRIMLEN ..... Returns the length of a character argument without counting trailing blanks 379
tsecnd, TSECND ..... Returns elapsed CPU time for a calling task during a multitasked program 381
unpack, UNPACK ..... Expands stored data 383

PVP Pages

auxstat, AUXSTAT ..... Gives the number of reads and writes to SDS that have occurred while accessing auxiliary array variables 385
bitvec, BITVEC ..... Generates a bit mask corresponding to integer or real arrays according to a specified condition 386
bitvecm, BITVECM ..... Generates a bit mask corresponding to masked integer arrays according to a specified condition 389
byt, PUTBYT, IGTBYT ..... Replaces a byte in a variable or an array 392
dtts, DTTS ..... Converts ASCII date and time to time stamp 393
icpused, ICPUSED ..... Gets CPU time in real-time clock (RTC) ticks 394
mtts, MTTS ..... Converts machine time (real-time clock value) to time stamp 395

mvc, MVC ..... Moves characters from one memory area to another 396
p32, P32, U32 ..... Packs or unpacks 32-bit words into or from Cray 64-bit words 397
p6460, P6460, U6064 ..... Packs or unpacks 60-bit words into or from Cray 64-bit words 398
second, SECOND ..... Returns elapsed CPU time 399
tremain, TREMAIN ..... Returns the CPU time (in seconds of type real) remaining for the program 402
tsdt, TSDT ..... Converts time stamps to ASCII date and time strings 403
tsmt, TSMT ..... Converts time stamp to machine time (real-time clock value) 404
unitts, UNITTS ..... Returns time-stamp units in specified standard-time units 405

NAME

INTRO_APPLIBS – Introduction to CrayLibs Application Libraries

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This manual describes different routines provided in the CrayLibs Application Libraries, which includes `libsma`, `libf`, and `libu`. A *library* is a collection of subroutines usually organized around a specific subject such as I/O. Routines generated in the form of inline code are described in *CF77 Commands and Directives*, publication SR–3771, or *CF90 Commands and Directives Reference Manual*, publication SR–3901.

The printed versions of the routines appear in two volumes and are grouped according to topics. Within each topic section, routines are further grouped into the following categories:

- Common routines which run on all Cray Research systems
- Routines that run only on Cray MPP systems
- Routines that run only on Cray PVP systems

Volume 1 contains the following topic sections:

- Calling Sequence: Calling sequences for Cray PVP systems.
- I/O Routines: File positioning, auxiliary `NAMELIST`, logical record, random-access file, asynchronous queued I/O, output suppression, and BOV/EOV tape processing.
- FFIO Routines: File opening, closing, read, and write for files using flexible file I/O (FFIO).
- SHMEM: Shared memory routines.
- Synchronization Routines: Task, lock, event, and barrier routines.
- Specialized Libraries: Character, packing, timing, and byte and bit manipulation routines.
- Appendix A: C Library Interface Routines: C routines and system calls that provide a C interface to Fortran programmers.

Volume 2 contains the following topic sections:

- Conversion Routines: Foreign file conversion (IBM, CDC, and VAX), numeric conversion, ASCII conversion, IEEE conversion, and other conversion routines.
- Heap, Table, and SDS Management Routines: Routines for manipulating and managing memory within heaps, tables, and secondary data segments (SDS).
- Programming Aid Routines: Flowtrace, traceback, dump, exchange package processing, and signal routines.
- System Interface Routines: Job control, floating-point interrupt, and special purpose interface routines.
- Search Routines: Search options to find maximum and minimum elements in vectors.

SUBPROGRAM CLASSIFICATION

Unless otherwise noted, all routines in this manual are described as Fortran or C subroutines or functions. In some cases (for example, `SECOND`), the routine may be called as either a subroutine or a function. The Cray Research CF77 compiling system enforces consistency in any one compilation unit. Programs written in C can call library functions intended for use by Fortran programs. C programmers are responsible for passing arguments by address and not by value, which is the normal case in C.

C programs can also be written to accommodate Fortran users. Such programs must be written to accept arguments passed by address rather than by value, which is the normal case in C. The program names must be in uppercase.

Pascal programs can call library functions intended for use by Fortran programs. Similarly, Fortran codes can invoke subroutines and functions written in Pascal. Unlike C, the Pascal compiler passes all arguments by address, and it supports several predefined conversion functions to facilitate communication with Fortran routines. See the *Pascal Reference Manual*, publication SR–0060, for information regarding parameter passing, data formats, and restrictions.

For more information on interlanguage calls, see *Interlanguage Programming Conventions*, publication SN–3009.

LIBRARY USAGE AND `segldr`

If you have created a routine with the same name as a default routine (one described in this manual or in your Fortran reference manual), your routine will be loaded if you specify it on, for instance, a `segldr`(1) command.

The segldr loader reads through the libraries completely, saving external information about each subroutine. User libraries are processed in the order specified to segldr, and the system libraries are processed next. Any subroutines required by the program are loaded with it.

NOTES

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is kind=8 for integer, real, complex and logical arguments.

SEE ALSO

INTRO_CONVERSION(3F), INTRO_FFIO(3F), INTRO_HEAP(3F), INTRO_INTERFACE(3F), INTRO_IO(3F), INTRO_PROGAIDS(3F), INTRO_SHMEM(3), INTRO_SORTSEARCH(3F), INTRO_SPECIALIBS(3F), INTRO_SYNC(3F)

NAME

`callseq` – Details the calling sequence for Cray PVP systems

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

This section details how subprograms are interlinked while running on Cray PVP systems. A subprogram corresponds to a function in C, a PROGRAM, SUBROUTINE or FUNCTION in Fortran, or a set of ENTER/EXIT macros in CAL.

Frame

A *frame* is a contiguous region of memory associated with a subprogram. At a minimum, a frame contains a pointer to the Traceback Name Block (TNB), the address of the caller (the return address, register B00), the address of the argument list (register B01), and the address of the caller's frame (register B02).

All subprograms have a frame except certain low-level (BASELVL) routines (see following).

In a stack environment (the default) the frame is allocated on the stack when the subprogram is invoked and the memory is released for re-use when the subprogram exits. A stack frame (see figure 1) will also contain storage for local variables and argument lists. For subprograms that are compiled with a static option, the frame is allocated at compile-time and is never released. Local variables and argument lists are also allocated at compile-time, though not as part of the frame.

Static and stack subprograms may call each other, but a static routine should not be called recursively nor should it be used in a multitasking environment.

Stack

The stack is created prior to entering the main program. The stack is implemented as one or more blocks (called *stack segments*) of memory in the user heap. The B66 register points to the current top of the stack; the B67 register points to the end of the current stack segment. As subprograms are invoked, their stack frame is created by incrementing the top of the stack by the size of the stack frame. When a stack frame cannot fit in the current segment this is called a stack overflow, and the stack manager is called to extend the segment, if possible, or to allocate a new segment from the heap. A link is established between the old and new segments and processing continues. When exiting subprograms, the top of the stack is decremented by the size of the stack frame. When a stack segment is empty, it may be returned to the heap.

The stack manager maintains a 128-word pad area beyond the end of the stack within the current stack segment. This is done so that subprograms can save B and T registers prior to checking for a segment overflow. If the stack manager must allocate a new segment, the saved B and T registers will be copied to the new segment. If the stack manager cannot extend a segment nor allocate a new segment to contain a stack frame, the program terminates with a stack allocation failure.

Traceback

The traceback information is a statically allocated block called the Traceback Name Block (TNB) which is generated at compile-time. The TNB contains information about each subprogram including its name, the address of its entry point, and information about the contents of the frame. The subprogram entry sequence puts the address of the TNB into the first word of the frame (non-BASELVL routines only). The TNB information can be used by run-time debuggers or to print out a traceback in the event of an abnormal program termination.

Subprogram Arguments

Most subprograms pass their arguments (both call-by-address and call-by-value arguments) in consecutive memory locations called an argument list. A one-word Argument List Header (ALH) is added to the beginning of the argument list. The ALH contains the number of words of arguments (for use by subprograms which accept a variable number of arguments) and the line number of the caller (for use by traceback and run-time debugging).

Certain intrinsic functions and VFUNCTIONS do not use an argument list. Instead, they pass arguments (usually call-by-value) through registers. These subprograms can accept only a limited number of arguments and do not have access to an ALH. By convention, these subprograms have a percent (%) as the last character of their name. The arguments are passed through consecutive S registers starting with S1. Note that each multi-word call-by-value argument requires the corresponding number of registers (for example, a complex value requires two registers). Vectorized versions of these functions are usually available. By convention, these vectorized subprograms have a percent (%) has the first and last character of their name. The arguments to vectorized functions are passed through the V registers in a manner analogous to the scalar functions. In addition, the VL register contains the vector length.

Function Results

Subprograms expect one-word function results (integers, pointers, floating-point numbers, and so forth) to be in the S1 register; two-word function results (complex or double-precision) to be in S1 and S2; and a double-precision complex result is returned in S1 through S4.

Languages which support functions that return aggregates (arrays, structures, and so forth) will pass an "extra" argument to the function containing the address of the aggregate.

Register Conventions

All registers are considered *saved*, *scratch*, or *reserved*, as follows:

- Saved registers must be saved and restored if used by the subprogram.
- Scratch registers may be used by the subprogram without saving and restoring them.
- Reserved registers must not be used by the subprogram except as outlined below.

The A, S, V, VL, and VM registers are scratch registers. The VL register is also an argument register for vectorized subprograms where it contains the vector length, but in those subprograms it can otherwise be treated as a scratch register.

Most of the B and T registers are saved registers. Subprogram linkage information is passed through B registers (B00 through B02) which are also saved registers. The stack pointers (B66 through B67) are reserved registers and are altered only during subprogram entry/exit.

Register	Status	Purpose
B00	saved	Return address (parcel form)
B01	saved	Pointer to the argument list
B02	saved	Pointer to the caller’s frame

Register	Status	Purpose
B03 through B65	saved	Saved B registers
B66	reserved	Pointer to the top of the stack
B67	reserved	Stack segment limit pointer
B70 through B77	scratch	Scratch B registers
T00 through T67	saved	Saved T registers
T70 through T77	scratch	Scratch T registers

It is important that the saved registers be stored in the proper place in the frame so that traceback, longjump, and so forth can locate the necessary information.

All of the shared B and T registers are considered scratch registers in non-multitasked programs. In multitasked programs, SB0 through SB3 and ST0 through ST3 are reserved for use by the multitasking library and should not be altered; the remaining shared B and T registers are available to the user.

Semaphores SM0 through SM15 are reserved for the multitasking and other run-time libraries and should not be altered. The remaining semaphores are available to the user.

The matrix in the Bit Matrix Multiply (BMM) functional unit (when present) is considered a scratch register.

BASELVL subprograms

Certain subprograms may utilize a light-weight entry/exit sequence (called BASELVL) to obtain better performance. However, these subprograms must operate under certain constraints:

- They must be leaf routines; that is, they cannot call other subprograms.
- They must not declare any local (stack) variables.
- They do not have a frame, thus certain traceback/debugging information must be kept in specific registers. These registers are considered reserved while executing a BASELVL subprogram.

Register	Purpose
B00	Return address (parcel form)
B01	-1 (indicates a BASELVL routine)
B77	TNB pointer
A1	Callers argument list pointer

Because BASELVL subprograms do not have a frame, they can execute their entry/exit sequence without memory references.

Argument List Header

The following table describes the Argument List Header

Field	Word	Bits	Description
ARFLAG	0	0	Nonstandard calling sequence flag (unused)
ARVAL	0	1	0 ≥ call-by-address, 1 ≥ call-by-value
ARCHR	0	2	0 ≥ non-character function; 1 ≥ char. function
ARCHK	0	3	Runtime argument checking flag (1 = on)

TRACEBACK Name Block

The following table describes the Traceback Name Block.

The following table describes the traceback information added before the entry point.

Field	Word	Bits	Description
TNBNN	−1	0−63	Last (or only) 8 characters of name.
TNBO	0	0−31	Mandatory zero.
TNBL	0	32−47	Number of following words (excluding this one).
TNBLN	0	48−63	Number of characters in the name. The first character of the name is contained in the -1 word −ICEIL(TNBLN,8) word of the table.
TNBPA	1	0−63	Parcel address of entry point.
TNBBL	2	0	Base level flag; set if this subprogram makes no calls.
TNBLT	2	1−7	Language type: 0 = cft or cft2 1 = PCC 2.0 2 = Pascal 3 = cft77 4 = CAL 5 = CFT 90 6 = C++

Field	Word	Bits	Description
			7 = Reserved 8 = SCC (ANSI C) 9 = Ada 10 = Lisp 11 = PCC
TNBCR	2	8–19	Maximum size of parameter list built by this subprogram.
TNBTVS	2	32–63	Size of stack temporary variable store.
TNBSCS	3	0–31	Size of static constant space.
TNBSVS	3	32–63	Size of static variable space.
TNBNT	4	50–56	Number of T registers.
TNBNB	4	57–63	Number of B registers.
TNBPS	5	0–63	Language-specific information; otherwise unspecified.

EXAMPLES

Figure 1 shows the relationship between stack segments after a stack overflow.

Include art, stored in Interleaf: 655F/department/art/tins/SR2165/fig1

NAME

INTRO_IO – Introduction to I/O routines

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The I/O routines include the following:

- Assign processing routines
- File positioning routines
- Auxiliary NAMELIST routines
- Logical record I/O routines
- Random-access file I/O routines
- Asynchronous queued I/O routines
- Fortran-callable tape processing routines involving beginning- and end-of-volume processing
- Miscellaneous I/O routines

Routines and Supported Machine Types

In the printed version of these routines, the man pages are grouped according to architecture type. The following list briefly describes this grouping:

- Common routines which run on all Cray Research systems:

AQCLOSE	AQOPEN
AQREAD	AQRECALL
AQSTAT	AQWAIT
AQWRITE	ASNCTL
ASNQFILE	ASSIGN
ASYNCMS	CHECKMS
CLOSEV	CLOSMS
FINDMS	FLUSH
FSUP	GETTP
GETWA	GTSTDPTR
NUMBLKS	OPENMS
PUTWA	READ

READC READMS
STINDX SYNCMS
WAITMS WCLOSE
WOPEN WRITE
WRITEC WRITMS

- Routines which run only on Cray PVP systems:
  ACPTBAD CHECKTP
  ENDSP READIBM
  RNL RNLECHO
  RNLSKIP RNLTYPE
  SETSP SETTP
  SKIPBAD SKIPF
  STARTSP TSYNC
  WNL WNLLINE
  WNLLONG WRITIBM
- Routines which run only on Cray MPP systems:
  none

The following routines run on SPARC systems: ASSIGN, ASNCTL, NUMBLKS, FLUSH

ASSIGN PROCESSING ROUTINES

Assign processing routines provide a library interface to assign processing.

The following routines are available for assign processing:

- ASNCTL(3F): controls functioning of ASSIGN, ASUNIT, ASNFILE, and ASNRM
- ASSIGN(3F): provides interface to assign processing from Fortran
- ASNUNIT(3F): assigns attributes to units (documented on the ASSIGN(3F) man page)
- ASNFILE(3F): assigns attributes to files (documented on the ASSIGN(3F) man page)
- ASNRM(3F): removes all entries in the assign environment (documented on the ASSIGN(3F) man page)
- ASNQFILE(3F): returns attributes for a file
- ASNQUNIT(3F): returns attributes for a unit (documented on the ASNQFILE(3F) man page)

FILE POSITIONING ROUTINES

File positioning routines change or indicate the position of the current file. These routines set the current positioning direction to input (read). If the previous processing direction is output (write), end-of-data is written on a blocked sequential file, and the buffer may be flushed. On a random file, the buffer is flushed.

The following routines are used for positioning:

- GETPOS(3I): returns current position of tape or mass storage file
- SETPOS(3I): returns to the position retained from the GETPOS request (documented on the GETPOS(3I) man page)
- GETTP(3F): receives position information about an opened tape file
- SETTP(3F): positions a specified tape file at a tape block
- SKIPF(3F): skips files

AUXILIARY NAMELIST ROUTINES

(Deferred implementation for Cray MPP systems.) NAMELIST routines let you control input and output defaults. No arguments are returned. For a more complete description of the NAMELIST feature, see *CF77 Commands and Directives*, publication SR–3771, or *CF90 Commands and Directives Reference Manual*, publication SR–3901.

The following routines are used for auxiliary NAMELIST routines:

- RNLCOMM(3F): deletes or adds a trailing comment indicator (documented on the rnl(3F) man page)
- RNLDELM(3F): deletes or adds a delimiting character (documented on the rnl(3F) man page)
- RNLFLAG(3F): deletes or adds an echo character (documented on the rnl(3F) man page)
- RNLREP(3F): deletes or adds a replacement character (documented on the rnl(3F) man page)
- RNLSEP(3F): deletes or adds a separator character (documented on the rnl(3F) man page)
- RNLECHO(3F): specifies the output unit for error messages and echo lines
- RNLSKIP(3F): takes action when an undesired NAMELIST group is encountered

- RNLTYPE(3F): determines the action if a type mismatch occurs across the equal sign on an input record
- WNLDELM(3F): defines an ASCII NAMELIST delimiter (documented on the wnl(3F) man page)
- WNLFLAG(3F): indicates the first ASCII character of the first line (documented on the wnl(3F) man page)
- WNLREP(3F): defines an ASCII NAMELIST separator (documented on the wnl(3F) man page)
- WNLLINE(3F): allows each NAMELIST variable to begin on a new line
- WNLLONG(3F): indicates output line length

LOGICAL RECORD I/O ROUTINES

The logical record I/O routines are divided into read routines, write routines, and bad data error recovery routines. The following routines are used for logical record I/O:

- ACPTBAD(3F): makes bad data available
- READ(3F): reads words, full record mode
- READP(3F): reads words, partial record mode (documented on the READ(3F) man page)
- READC(3F): reads characters, full record mode
- READCP(3F): reads characters, partial record mode (documented on the READC(3F) man page)
- READIBM(3F): reads two IBM 32-bit real number words from each Cray 64-bit word
- SKIPBAD(3F): skips bad data
- WRITE(3F): writes words, full record mode
- WRITEP(3F): writes words, partial record mode (documented on the WRITE(3F) man page)
- WRITEC(3F): writes characters, full record mode
- WRITECP(3F): writes characters, partial record mode (documented on the WRITEC(3F) man page)
- WRITIBM(3F): writes two IBM 32-bit real number words from each Cray 64-bit number word

ACPTBAD(3F), READIBM(3F), SKIPBAD(3F), and WRITIBM(3F) are not supported on Cray MPP systems.

Read Routines

Read routines transfer partial or full records of data from the external device to the user data area. Depending on the read request issued, the data is placed in the user data area either 1 character per word or in full words. In partial mode, the file maintains its position after the read is executed. In record mode, the file position is maintained after the end-of-record (EOR) that terminates the current record. Operation of these routines depends on the -F and -s options on the assign(1) command.

Write Routines

Write routines transfer partial or full records of data from the user data area to the external device. Depending on the write operation requested, data is either taken from the user data area 1 character per word and packed 8 characters per word or is transferred in full words. In partial mode, no end-of-record (EOR) is inserted in the external device in the word following the data that terminates the record. Operation of these routines depends on the -F and -s options on the assign(1) command.

Bad Data Error Recovery Routines

Bad data error recovery routines enable a user program to continue processing a file when bad data is encountered. *Bad data* refers to an unrecovered error encountered while the file was being read. Skipping the data forces the file to a position past the bad data, so that no data is transferred to the user-specified buffer. Accepting the data causes the bad data to be transferred to a user-specified buffer. The file is then positioned immediately following the bad data.

When an unrecovered data error is encountered, continue processing by calling either the SKIPBAD or the ACPTBAD routine.

These routines do not work with foreign data conversion.

RANDOM-ACCESS FILE I/O ROUTINES

Sequentially accessed files are used for applications that read input only once during a process and write output only once during a process. However, when large numbers of intermediate results are used randomly as input at various stages of programs, a random-access file capability is more efficient than sequential access. A random-access file consists of records that are accessed and changed. Random access of data eliminates the slow processing and inconvenience of sequential access when the order of reading and writing records differs in various applications.

Random-access file I/O routines let you specify how records of a file are to be changed, without the usual limitations of sequential access. Choose specific routines based on performance requirements and the type of access needed.

Random-access files can be created and accessed by the record-addressable, random-access file routines (READMS/WRITMS and READDR/WRITDR) or the word-addressable, random-access file routines (GETWA/PUTWA).

Note: Files created by this method are distinct from Fortran direct-access I/O files. Also, random-access file I/O routines used in a program with segmented programs should reside in the root segment. However, if all I/O is done within one segment, the routines can reside in that segment.

Record-addressable, Random-access File I/O Routines

Record-addressable, random-access file I/O routines let you generate files containing variable-length, individually addressable records. These records can be read and rewritten at your discretion. The library routines update indexes and pointers. The random-access file information is stored in two places: in an array in user memory and at the end of the random-access file.

When a random-access file is opened, the OPEN routine initializes an array in user memory to contain the master index to the records of the file. This master index contains the pointers to and, optionally, the names of the records within the file. Although you provide this storage area, it must be modified only by the random-access file I/O routines.

When a random-access file is closed and optionally saved, the storage area containing the master index is rewritten at the end of the random-access file, thus recording changes to the contents of the file.

The following Fortran-callable routines can change or access a record-addressable, random-access file: OPENMS, WRITMS, READMS, CLOSMS, FINDMS, CHECKMS, WAITMS, ASYNCMS, SYNCMS, OPENDR, WRITDR, READDR, CLOSDR, STINDR, CHECKDR, WAITDR, ASYNCDR, SYNCDR, and STINDX.

The READDR/WRITDR random-access I/O routines are direct-to-disk versions of READMS/WRITMS. All input or output goes directly between the user data area and the mass storage file without passing through a system-maintained buffer. Because mass storage can be addressed only in 512-word blocks, all record lengths are rounded up to the next multiple of 512 words.

You can intermix READMS/WRITMS and READDR/WRITDR files in the same program, but you must not use the same file in both packages simultaneously.

OPENMS/OPENDR opens a file and specifies the file as a random-access file that can be accessed or changed by the record-addressable, random-access file I/O routines. If the file does not exist, the master index contains zeros; if the file does exist, the master index is read from the file. The master index contains the current index to the file. The current index is updated when the file is closed using CLOSMS/CLOSDR.

A single program can use up to 40 active READMS/WRITMS files and 20 READDR/WRITDR files.

The following routines are used for record-addressable, random-access I/O:

- ASYNCMS(3F), ASYNCDR(3F): sets the I/O mode to be asynchronous
- CHECKMS(3F), CHECKDR(3F): checks the status of an asynchronous I/O operation
- CLOSMS(3F), CLOSDR(3F): closes a random-access file and writes the master index
- FINDMS(3F): reads records into data buffers used by random-access file routines
- OPENMS(3F), OPENDR(3F): opens a local file as a random-access file
- READMS(3F), READDR(3F): reads a record from a random-access file to memory
- STINDX(3F), STINDR(3F): allows an index to be used as the current index by creating a subindex
- SYNCMS(3F), SYNCDR(3F): sets the I/O mode to be synchronous
- WAITMS(3F), WAITDR(3F): waits for completion of an asynchronous I/O operation
- WRITMS(3F), WRITDR(3F): writes data from user memory to a random-access file and updates the index

Word-addressable, Random-access File I/O Routines

A word-addressable, random-access file consists of an adjustable number of contiguous words. You can access any word or contiguous sequence of words from a word-addressable, random-access file by using the associated routines. These files and their I/O routines are similar to the record-addressable, random-access files and their routines, except that an index is not meaningful. The Fortran-callable, word-addressable random-access I/O routines are WOPEN, WCLOSE, PUTWA, APUTWA, GETWA, and SEEK. WOPEN opens a file and specifies it as a word-addressable, random-access file that can be accessed or changed with the word-addressable routines. The WOPEN call is optional. If a call to GETWA or PUTWA is executed first, the file is opened for you with the default number of blocks (16), and *istats* is turned on.

The following routines are used for word-addressable, random-access file I/O:

- GETWA(3F): synchronously reads words from the file into user memory
- SEEK(3F): asynchronously reads data into file buffers (documented on the GETWA(3F) man page)
- PUTWA(3F): synchronously writes words from memory to the file
- APUTWA(3F): asynchronously writes words from memory to the file (documented on the PUTWA(3F) man page)
- WCLOSE(3F): finalizes additions and changes and closes the file
- WOPEN(3F): opens a file and specifies it as a word-addressable, random-access file

ASYNCHRONOUS QUEUED I/O ROUTINES

Asynchronous queued I/O routines initiate a transfer of data and allow the subsequent execution sequence to proceed concurrently with the actual transfer.

The following routines are used for asynchronous queued I/O:

- AQCLOSE(3F): closes an asynchronous queued I/O file
- AQOPEN(3F): opens a file for asynchronous queued I/O
- AQOPENM(3F): opens a file for asynchronous queued I/O using specified status flag (documented on the AQOPEN(3F) man page)
- AQREAD(3F): queues a simple asynchronous I/O read request
- AQREADC(3F): queues a compound asynchronous I/O read request (documented on the AQREAD(3F) man page)
- AQRECALL(3F): waits for completion of a particular set of asynchronous queued I/O requests
- AQSTAT(3F): checks the status of asynchronous queued I/O requests
- AQWAIT(3F): waits for completion of all asynchronous queued I/O requests
- AQWRITE(3F): queues a simple asynchronous I/O write request
- AQWRITEC(3F): queues a compound asynchronous I/O write request (documented on the AQWRITE(3F) man page)

FORTRAN-CALLABLE TAPE PROCESSING ROUTINES

(Deferred implementation for Cray MPP systems.) Fortran-callable routines perform special functions on a tape file.

- `CHECKTP`(3F): checks tape position
- `CLOSEV`(3F): closes volume and mounts next volume in Volume Identifier list
- `ENDSP`(3F): disables special tape processing
- `SETSP`(3F): enables and disables EOV processing
- `STARTSP`(3F): enables special tape processing

MISCELLANEOUS I/O ROUTINES

`FLUSH` writes to a file any buffered data previously written by Fortran output statements.

`GTSTDPTR` maps a standard file descriptor into the corresponding address of the `FILE` structure (stream pointer) for subsequent use in the Fortran-callable `fread`(3C) and `fwrite`(3C) routines.

The `LENGTH` and `UNIT` routines allow a Fortran program to issue a wait until a `BUFFER IN` or `BUFFER OUT` operation has completed. This type of wait is crucial if the program must synchronize a data transfer with further use of the information being transferred.

The `NUMBLKS` function returns the current file size of a specified unit or file.

The `UNITS` function waits for completion of a `BUFFER IN` or `BUFFER OUT` operation and returns the status of that operation.

FORTRAN I/O UNITS

Cray Research Fortran implementations under the UNICOS operating system allow I/O units to be connected to files, and they serve as a means of referring to that file. This subsection describes the characteristics of those unit identifiers.

A Fortran unit identifier is one of the following:

- An integer variable or expression with a 0 or positive value. Each integer unit identifier (*i*) is associated with the file `fort.`*i*, which may exist. For example, unit 10 is associated with the file `fort.10` in the current working directory.

- An asterisk, identifying a particular file that is connected for formatted, sequential access (allowed only on READ and WRITE statements). On READ statements, an asterisk refers to unit 100 (standard input). On WRITE statements, an asterisk refers to unit 101 (standard output).
- A Hollerith (integer) variable consisting of 1 to 7 left-justified, zero-filled ASCII characters. Each Hollerith unit identifier is associated with the file of the same name, which may or may not exist. For example, unit 'red'L is associated with the file red in the current working directory.

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

Fortran INQUIRE and CLOSE statements may refer to any valid or invalid unit number. All other Fortran I/O statements may refer only to valid unit numbers. For the purposes of an executing Fortran program, all unit numbers in use or available for use by that program are valid (that is, they exist). All unit numbers that are not available for use are invalid (that is, they do not exist).

Valid unit numbers are 0 through 299. Unit numbers 103 and 104 are reserved. Unit numbers 0, 5, and 6 are associated with the special UNICOS files standard error, standard input, and standard output, respectively. All other valid unit numbers are associated with the file fort.*i*, as previously indicated. Some additional valid unit numbers may, as a site option, be available, starting at number 300. Use the INQUIRE statement to check the validity (existence) of any unit number prior to using it. Example:

```
LOGICAL UNITOK, UNITOP
...
INQUIRE (UNIT = I, EXIST = UNITOK, OPENED = UNITOP)
IF (UNITOK .AND. .NOT. UNITOP) THEN
  OPEN (UNIT = I, ...)
ENDIF
```

Initially, all valid units are closed. A unit is connected to a file as the result of one of three types of opens:

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

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

  If unit number 4 is not open, the WRITE statement will cause it to be connected to the associated file (fort.4), unless overridden by an assign(1) command referencing unit 4.

- An explicit unnamed open occurs when the first reference to a unit number is an OPEN statement without a FILE specifier. Example:

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

  If unit number 7 is not open, the OPEN statement will cause it to be connected to the associated file (fort.7), unless overridden by an assign command referencing unit 7.

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

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

  If unit number 9 is not open, the OPEN statement will cause it to be connected to the file blue, unless overridden by an assign command referencing the file blue.

The BACKSPACE, ENDFILE, and REWIND statements will not do an implicit open. If the unit is not connected to a file, the requested operation is disregarded.

Unit numbers 100 through 102 are permanently associated with the special UNICOS files standard input, standard output, and standard error, respectively. They may be referenced on READ and WRITE statements. A CLOSE statement on these unit numbers has no effect. An INQUIRE statement on these unit numbers will indicate they are nonexistent (invalid). These unit numbers exist to allow guaranteed access to the special UNICOS files, standard input, standard output, and standard error, without regard to any unit actions taken by an executing program. Thus, a READ or WRITE I/O statement with an asterisk unit identifier (or a PRINT statement) will always work. Nonstandard I/O operations on these units (for example, BUFFER IN, BUFFER OUT, READMS, WRITEMS, and so on) are not supported.

Fortran application or library subroutines that want to access the special UNICOS files will have that access by using unit numbers 100 through 102, even if the user program closes or reuses unit numbers 0, 5, and 6.

Unit numbers 103 and 104 are reserved for Cray Research. A CLOSE statement on these unit numbers has no effect. An INQUIRE statement on these unit numbers will indicate they are nonexistent. They may not be used in any other Fortran I/O statement.

Special Files

For all unit numbers associated with the special UNICOS files standard input, standard output, and standard error, the nature of that association is sequential and formatted. That is, unformatted and/or direct-access statements are not allowed on those files. Furthermore, the special UNICOS file standard input is read-only. `ENDFILE`, `PRINT`, and `WRITE` statements are not allowed, and the special UNICOS files standard error and standard output are write-only (`READ` statements are not allowed).

`REWIND` and `BACKSPACE` statements are permitted on terminal files, but with no effect; `ENDFILE` is permitted on terminal files, unless they are read-only (standard input). It writes a logical endfile record.

For all unit numbers associated with pipes, the `REWIND` statement is invalid. If the file logically has to be repositioned, the `BACKSPACE` statement is invalid. A situation in which the file may not have to be positioned is a `BACKSPACE` over an endfile record for a file that has only a logical representation of an endfile record (for example, text files).

Inquire-by-Unit Processing

The `INQUIRE` statement can be used to retrieve the name of a file connected to a specified unit, if the file has a name. The FORTRAN 77 standard allows for unnamed files.

The special UNICOS files standard input, standard output, and standard error are unnamed, and an `INQUIRE` on a unit connected to any of these files indicates that it is unnamed (the `NAMED` parameter is set to `.FALSE.`).

An `INQUIRE` on any unit connected using an explicit named open will indicate that the file is named and will return as the name, the name that was given on the `FILE` specifier on the `OPEN` statement.

An `INQUIRE` on any unit connected using an explicit unnamed open or an implicit open, may or may not indicate that the file is named. A name will be returned only if the Fortran run-time library can ensure that a subsequent open, with the `FILE` specified as the name, will connect to the same file. For scratch files (files opened with `STATUS='SCRATCH'`), a name is returned so the same `assign`(1) options associated with the file name are used.

CAUTIONS

The property of unit numbers 100 through 102, whereby they appear nonexistent or invalid on an `INQUIRE` or `OPEN` and yet I/O is permitted, is a Cray Research extension to the FORTRAN 77 standard.

It is possible that the UNICOS special files (standard input, standard output, and standard error) can have two different units connected to them simultaneously (units 0/102, 5/100, and 6/101). This is a Cray Research extension to the FORTRAN 77 standard.

SEE ALSO

For more information about Fortran I/O units, see the following:

The `assign`(1) man page in the *UNICOS User Commands Reference Manual*, publication SR–2011

For more information on I/O for Fortran programmers, see the *Application Programmer's I/O Guide*, publication SG–2168. This guide describes the type of I/O available and provides insight into the efficiencies and inefficiencies of each. It demonstrates how to speed up various forms of I/O and describes tools that extract statistics from the execution of a Fortran program.

Among the types described are standard Fortran I/O, system I/O, and I/O extensions to Fortran. Also described is `assign`(1), file structures, buffering, flexible file I/O, and devices. This should give you the information necessary to make informed decisions on effective and efficient use of I/O on Cray Research systems.

NAME

AQCLOSE – Closes an asynchronous queued I/O file

SYNOPSIS

CALL AQCLOSE(*aqp*, *status* [,*reply*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQCLOSE closes a file opened by using AQOPEN(3F). AQCLOSE starts all queued AQIO requests, waits for their completion, and then closes the file. If any queued or busy requests have pending error conditions, an error status is returned, and *reply* (if present) is set to the request ID of the failed request.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer variable or array element. The name of the array in the user's program that contains the asynchronous queued I/O file handle. This must be the same variable specified in the AQOPEN request.

status Integer variable. Status code *status* returns any errors or status information to the user. On output from AQCLOSE, *status* has one of the following values:

- =0 No error detected.
- <0 Error detected. The absolute value of *status* indicates the cause of the error. See intro(2) and the Fortran library error messages. The conditions that may cause AQCLOSE to return an error status include the close(2) of the file failed; the *aqp* argument was invalid; a previously asynchronous I/O request to this file failed, and the resulting error status had not previously been returned to the program.

reply Optional integer variable. If present, and if an error is due to a failed I/O request, it returns the request ID of the most recent failed request.

SEE ALSO

AQOPEN(3F), AQREAD(3F), AQREADC (see AQREAD(3F)), AQRECALL(3F), AQSTAT(3F), AQWAIT(3F), AQWRITE(3F), AQWRITEC (see AQWRITE(3F))

Application Programmer's I/O Guide, publication SG–2168

NAME

AQOPEN, ASOPENM – Opens a file for asynchronous queued I/O

SYNOPSIS

```
CALL AQOPEN(aqp, aqpsize, dn, status)
CALL AQOPENM(aqp, aqpsize, name, oflag, status)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQOPEN opens a file to be used in subsequent AQIO requests, such as AQREAD(3F) or AQWRITE(3F).

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments. The following is a list of valid arguments for this routine.

aqp Integer variable or array element that will be assigned the asynchronous queued I/O file handle.

aqpsize Integer variable, expression, or constant. Maximum number of queued I/O requests.

dn A Fortran character constant or variable containing the name of the file. An integer variable, expression, or constant. *dn* can be a unit number or the name of the file as an L-type Hollerith constant.

name A Fortran character constant or variable containing the name of the file.

oflag An integer input variable. *oflag* contains the status flags that AQOPENM passes to the open(2) system call when opening the specified file. The value for *oflag* may be obtained from IPXFCONST. For more information about the possible values, see the open(2) man page. AQOPENM does not verify the validity of the *oflag* specified. For the default file type, the status flag that AQOPEN uses includes the O_RAW bit; if you use AQOPENM, you must explicitly specify this if it is desired.

status Integer variable. Status code *status* returns any errors or status information to the user. On output from AQOPEN, *status* has one of the following values:

=0 No errors detected.

<0 Error detected. If the absolute value of *status* is ≥ 1000, it is a library error, and you can type `explain lib-x` to view an explanation. If the absolute value of *status* is < 1000, it is probably a system error; type `explain sys-x` to view an explanation.

Asynchronous queued I/O provides a method of random access to or from mass storage into buffers in user memory.

NOTES

A file opened using AQOPEN must be closed by AQCLOSE(3F). Failure to close the file in this way will cause unpredictable behavior.

It is not valid to mix calls to AQIO routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

AQCLOSE(3F), AQREAD(3F), AQRECALL(3F), AQSTAT(3F), AQWAIT(3F), AQWRITE(3F)

Application Programmer's I/O Guide, publication SG–2168

NAME

AQREAD, AQREADC – Queues a simple or compound asynchronous I/O read request

SYNOPSIS

CALL AQREAD(*aqp*, *cpuadd*, *blknum*, *blocks*, *reqid*, *queue*, *status*)

CALL AQREADC(*aqp*, *cpuadd*, *mstride*, *blknum*, *blocks*, *dstride*, *incs*, *reqid*, *queue*, *status*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQREAD and AQREADC transfer data between the data buffer and the device on which the file resides. Requests can be simple (AQREAD) or compound (AQREADC). A simple request is one in which data from consecutive sectors on the disk is read into one buffer. A compound request is one in which several simple requests are separated by a constant number of sectors on disk, a constant number of memory words for buffers, or both.

On the CRAY T3D system, AQREADC results in one or more I/O system calls. Each compound request is broken down into the simple requests that comprise it. The number of broken-down simple requests that are issued via a single system call is limited; this limit may be adjusted by setting the MAXAQCMPND environment variable to the desired value. This environment variable affects only compound requests.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer array. The name of the array in the user's program that contains the asynchronous queued I/O file handle. Must be the same array as specified in the AQOPEN(3F) request.

cpuadd Type determined by the user; this may not be a Fortran character variable. Starting memory address; the location where the first word of data is placed.

blknum Integer variable, expression, or constant. Starting block number. The block number of the first block to be read on this request. Block numbering starts at 0.

blocks Integer variable, expression, or constant. The number of 512-word blocks to be read.

reqid Integer variable, expression, or constant. A user-supplied value for identifying a particular request.

queue Integer variable, expression, or constant. Queue flag. If 0, I/O is initiated provided that I/O on the file is not already active. If the queue flag is set to nonzero, the request is added to the queue but no attempt is made to start I/O.

status Integer variable. Status code *status* returns any errors to the user. On output from these routines, *status* has one of the following values:

`=0` No error detected.

`<0` Error detected. The absolute value of *status* indicates the cause of the error. See `intro`(2) for more information. `AQREAD` and `AQREADC` may return an error status if a previous asynchronous I/O request to this file failed, and the resulting error status had not been returned to the program.

dstride Integer variable, expression, or constant. Disk stride; the number of disk blocks to skip between the base addresses of consecutive transfers. The stride value may be positive (to skip forward), negative (to skip backward), or 0. This parameter is valid only for compound requests.

incs Integer variable, expression, or constant. The number of simple requests minus 1 that comprise a compound request. Zero (0) implies a simple request. This parameter is valid only for compound requests.

mstride Integer variable, expression, or constant. Data buffer stride; the number of memory words to skip between the base addresses of consecutive transfers. The stride value may be positive (to skip forward), negative (to skip backward), or 0 (implying a contiguous data flow to memory). This parameter is valid only for compound read requests.

SEE ALSO

`AQCLOSE`(3F), `AQOPEN`(3F), `AQRECALL`(3F), `AQSTAT`(3F), `AQWAIT`(3F), `AQWRITE`(3F)

`intro`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

AQRECALL – Waits for completion of asynchronous queued I/O request

SYNOPSIS

CALL AQRECALL(*aqp, status, reqid, reply*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQRECALL leaves the program suspended until a particular asynchronous queued I/O request (and all the asynchronous I/O requests made with the same *aqp* that preceded it) is complete, or until an error is encountered.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer array. The name of the array in the user's program that contains the asynchronous queued I/O file handle. This must be the same array specified in the AQOPEN(3F) request.

status Integer variable. This status code returns one of the following values on output from AQRECALL:

- 0 The requested ID is not found in the list of I/O requests.
- +5 I/O has been completed.
- <0 A negative value indicates that the I/O request indicated by *reply* has failed. Control is returned immediately. The absolute value of the status returned indicates the reason. See intro(2) for more details.

reqid If nonzero, AQRECALL waits until the I/O identified by this value (and all that preceded it) are complete, or until an error is encountered.

If zero, AQRECALL waits until all outstanding I/O is complete, or until an error is encountered.

AQRECALL searches the queue of requests until it finds the first match for *reqid*. If more than one entry in the request queue has the same *reqid*, results are unpredictable. To effectively use AQRECALL, ensure that each entry in the queue has a unique request ID. Entries may remain in the queue after AQWAIT or AQRECALL.

reply Integer variable. If an error occurred, *reply* is set to the request ID of the entry that incurred the error. Control is returned immediately. If more than one outstanding request has a pending error, AQRECALL must be called repeatedly until the return status is 0 to call all error returns.

SEE ALSO

AQOPEN(3F), AQREAD(3F), AQSTAT(3F), AQWAIT(3F), AQWRITE(3F)

intro(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

AQSTAT – Checks the status of asynchronous queued I/O requests

SYNOPSIS

CALL AQSTAT(*aqp*, *reply*, *reqid*, *status* [,*wait*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQSTAT returns the status of a particular queued request. If the optional *wait* argument is specified, return is delayed until that request is complete.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer array. The name of the array in the user's program that contains the asynchronous queued I/O file handle. This must be the same array specified in the AQOPEN(3F) request.

reply Integer variable. Specify a dummy integer in this position because the routine expects four arguments in a particular order.

reqid Integer variable, expression, or constant. Specifies the request ID for which status information will be returned.

status Integer variable. This status code returns one of the following values on output from AQSTAT:

- 0 The requested ID is not found in the list of I/O requests. This can occur if the requested ID does not correspond to an actual request, or if the requested ID has completed and the entry has been overwritten.
- +2 The requested ID is in the library queue, but the system request to start this I/O has not been made.
- +4 The asynchronous request is queued for I/O in the operating system.
- +5 I/O has been completed.

`<0` A negative value indicates that this I/O request failed. The absolute value of the status returned indicates the reason. See `intro`(2) for more details.

wait Optional argument. If present and nonzero, return is delayed until the specified request is complete.

SEE ALSO

`AQCLOSE`(3F), `AQOPEN`(3F), `AQREAD`(3F), `AQRECALL`(3F), `AQWAIT`(3F), `AQWRITE`(3F)

`intro`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

AQWAIT – Waits for completion of asynchronous queued I/O requests

SYNOPSIS

CALL AQWAIT(*aqp*, *status* [,*reply*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQWAIT delays program execution until all pending requests have completed or until an error is encountered. If necessary, requests that have been queued but not started are started.

If any of the pending requests encounters an error, control is returned immediately, and the variable *reply*, if specified, is set to the request ID of the request that encountered the error.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer array. The name of the array in the user's program that contains the asynchronous queued I/O parameter block. This must be the same array specified in the AQOPEN(3F) request.

status Integer variable that returns any errors or status information to the user. On output from AQWAIT, *status* has one of the following values:

- 2 No I/O is active or queued.
- =0 No errors detected.
- <0 Error detected. The absolute value of the *status* indicates the cause of the error. See intro(2) for more information.

reply Optional argument. Request ID of request in error.

SEE ALSO

AQCLOSE(3F), AQOPEN(3F), AQREAD(3F), AQRECALL(3F), AQSTAT(3F), AQWRITE(3F)

intro(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

AQWRITE, AQWRITEC – Queues a simple or compound asynchronous I/O write request

SYNOPSIS

CALL AQWRITE(*aqp*, *cpuadd*, *blknum*, *blocks*, *reqid*, *queue*, *status*)

CALL AQWRITEC(*aqp*, *cpuadd*, *mstride*, *blknum*, *blocks*, *dstride*, *incs*, *reqid*, *queue*, *status*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

AQWRITE and AQWRITEC transfer data between the device on which the file resides and the data buffer. Requests can be simple (AQWRITE) or compound (AQWRITEC). A simple request is one in which data from one buffer is written to consecutive sectors on disk. A compound request is one in which several simple requests are separated by a constant number of sectors on disk, a constant number of memory words for buffers, or both.

On Cray MPP systems, AQWRITEC results in one or more I/O system calls. Each compound request is broken down into the simple requests that comprise it. The number of broken-down simple requests that are issued via a single system call is limited; this limit may be adjusted by setting the MAXAQCMPND environment variable to the desired value. This environment variable affects only compound requests.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

aqp Integer array. The name of the array in the user's program that contains the asynchronous queued I/O file handle. Must be the same array specified in the AQOPEN request.

cpuadd Type determined by user; this may not be a Fortran character variable. Starting memory address; the location of the first word in the user's program to be written.

blknum Integer variable, expression, or constant. The block number of the first block to be written on this request. Block numbering starts at 0.

blocks Integer variable, expression, or constant. The number of 512-word blocks to be written.

reqid Integer variable, expression, or constant. A user-supplied value for identifying a particular request.

queue Integer variable, expression, or constant. Queue flag. If 0, I/O is initiated when I/O on the file is not already active. If the queue flag is set to nonzero, the request is added to the queue but no attempt is made to start I/O.

status Integer variable. Status code *status* returns any errors to the user. On output from these routines, *status* has one of the following values:

- `=0` No error detected.
- `<0` Error detected. The absolute value of *status* indicates the cause of the error. See `intro`(2) for more information. `AQREAD` and `AQREADC` may return an error status if a previous asynchronous I/O request to this file failed, and the resulting error status had not been returned to the program.

mstride Integer variable, expression, or constant. Data buffer stride; the number of memory words to skip between the base addresses of consecutive transfers. The stride value may be positive (to skip forward), negative (to skip backward), or 0 (implying contiguous data flow from memory). This parameter is valid only for compound write requests.

dstride Integer variable, expression, or constant. Disk stride; the number of disk blocks to skip between the base addresses of consecutive transfers. The stride value may be positive (to skip forward), negative (to skip backward), or 0. This parameter is valid only for compound requests.

incs Integer variable, expression, or constant. The number of simple requests minus 1 that compose a compound request. Zero (0) implies a simple request. This parameter is valid only for compound requests.

SEE ALSO

`AQCLOSE`(3F), `AQOPEN`(3F), `AQREAD`(3F), `AQRECALL`(3F), `AQSTAT`(3F), `AQWAIT`(3F)

`intro`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

ASNCTL – Controls function of ASSIGN, ASNFILE, ASNUNIT, and ASNRM routines

SYNOPSIS

CALL ASNCTL(*option, value, ier*)

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

The 'LOCAL' and 'NEWLOCAL' modes are useful for any utility written in Fortran when you want to use ASSIGN(3F) but do not want to access the assign environment file set up by the user.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

option A Fortran character variable containing either of the following values:

´LOCAL´ Causes ASNCTL to establish a local assign environment. The old assign environment is copied into the newly created local assign environment. Subsequent calls to ASSIGN, ASNUNIT, ASNFILE, and ASNRM (see ASSIGN(3F)) affect only the newly created local assign environment. Any future Fortran OPEN requests will also read from the local assign environment initiated by this call to ASNCTL. The user must set *value* to 1 when calling ASNCTL with *option*='LOCAL'.

´NEWLOCAL´ Causes ASNCTL to establish an empty local assign environment. Subsequent calls to ASSIGN, ASNUNIT, ASNFILE, and ASNRM (see ASSIGN(3F)) affect the newly created local assign environment. Any future Fortran OPEN requests will also read from the local assign environment initiated by this call to ASNCTL. The user must set *value* to 1 when calling ASNCTL with *option*='NEWLOCAL'.

´RESTORE´ Restores the assign environment that was active before the preceding 'LOCAL' or 'NEWLOCAL' request. Sets *value* to 1 when calling ASNCTL with *option*='RESTORE'.

value Integer variable, constant, or array element containing the option value.

ier Integer variable or array element that is assigned the error status on return. Zero (0) is usually returned, indicating no errors were encountered; otherwise, a positive error status is returned.

EXAMPLES

The writer of the following Fortran program wants to disregard any assign information provided by the user. However, the program requires sequential unformatted I/O on unit 11 with an unblocked file structure. This is accomplished by the following:

```
CALL ASNCTL('NEWLOCAL',1,ier)       ! start local assign environment
CALL ASNUNIT(11,'-s unblocked',ier) ! assign the "unblocked" file structure
OPEN(11,form='unformatted')         ! open unit 11
```

SEE ALSO

ASSIGN(3F)

NAME

ASNQFILE, ASNQUNIT – Returns the assign options currently in effect for a file name or unit number

SYNOPSIS

CALL ASNQFILE(*file*, *attr*, *istat*)

CALL ASNQUNIT(*iun*, *attr*, *istat*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ASNQFILE queries the assign environment for any assign options currently in effect for a file name. ASNQUNIT queries the assign environment for any assign options currently in effect for a unit number. The assign options may have been established earlier by the assign(1) command or the ASSIGN, ASNUNIT, or ASNFILE (see ASSIGN(3F)) library routines.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

file A character variable or array element containing the file name.

attr A character variable or array element that receives the option's value in the form of a character string. If the options exceed the length of *attr*, *istat* is set to a positive error code. If no assign options are found for the specified unit or file, ' ' is returned in *attr*.

istat An integer variable or array element that receives one of the following return statuses:

- 0 Options were found for the specified *file* or *iun*
- −1 No options were found for the specified *file* or *iun*.
- >0 An error condition was encountered. Use the explain(1) command to obtain information about a particular error code.

iun An integer variable or array element containing the unit number.

SEE ALSO

ASNCTL(3F), ASSIGN(3F)

assign(1) in the *Application Programmer's I/O Guide*, publication SG–2168

explain(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

ASSIGN, ASNUNIT, ASNFILE, ASNRM – Provides library interface to assign processing

SYNOPSIS

```
CALL ASSIGN(cmd [,ier])
CALL ASNUNIT(iunit, astring, ier)
CALL ASNFILE(fname, astring, ier)
CALL ASNRM(ier)
```

Cray PVP systems and Cray MPP systems only:

```
CALL ASSIGN(cmd)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

ASSIGN provides an interface to assign processing from Fortran.

ASNUNIT and ASNFILE assign attributes to units and files, respectively.

ASNRM removes all entries currently in the assign environment.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

cmd A Fortran character variable containing a complete assign(1) command in the format also acceptable to ISHELL(3F). The –V option cannot be processed by the ASSIGN routine.

ier An integer variable that is assigned the exit status on return. A 0 indicates normal return; >0 indicates a specific error status

iunit An integer variable or constant containing the unit number to which attributes are assigned.

astring A Fortran character variable containing any attribute options and option values that could be passed to assign(1). Control options -I, -O, and -R can also be passed.

fname A character variable or constant containing the file name to which attributes are assigned.

NOTES

Users are encouraged to use the ASSIGN library routines rather than ISHELL('assign'), because ISHELL(3F) causes vfork(2), an exec(2) of sh(1), another fork(2), and another exec(2) of assign(1).

EXAMPLES

Example 1: The following is equivalent to assign -s unblocked f:file

```
CALL ASSIGN('assign -s unblocked f:file')
```

Example 2: The following has the same effect as assign -I -n 2 u:99

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

Example 3: The following is equivalent to executing assign -s tape u:1

```
CALL ASNUNIT(1,' -s tape',IER)
```

SEE ALSO

ASNCTL(3F), ASNQFILE(3F), ASNQUNIT(3F), ISHELL(3F)

assign(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

ASYNCMS, ASYNCDR – Sets I/O mode for random-access routines to asynchronous

SYNOPSIS

```
CALL ASYNCMS(dn [,ierr])
CALL ASYNCDR(dn [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

As the ASYNCMS or ASYNCDR routine sets the I/O mode for the random-access routines to be asynchronous, I/O operations can be initiated, and subsequent execution can proceed simultaneously with the actual data transfer. If using READMS(3F), precede asynchronous reads with calls to FINDMS(3F).

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

ierr Error control and code. Specify an integer variable. If you specified *ierr* on the call to ASYNCMS/ASYNCDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned, and the message is added to the program's stderr file. On output from ASYNCMS/ASYNCDR, *ierr* has one of the following values:

= 0 No errors detected.

≤ 0 Error detected. *ierr* contains one of the following error codes:

−1 The file name or unit number is not valid.

−15 OPENMS/OPENDR was not called on this file.

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

CHECKMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), READMS(3F), STINDX(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F)

NAME

CHECKMS, CHECKDR – Checks status of asynchronous random-access I/O operation

SYNOPSIS

```
CALL CHECKMS(dn, istat [,ierr])
CALL CHECKDR(dn, istat [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

CHECKMS and CHECKDR routines check the status of an asynchronous random-access I/O operation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn Integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

istat File I/O Activity flag. Specify an integer variable.

= 0 No I/O activity on the specified file

= 1 I/O activity on the specified file

ierr Error control and code. Specify an integer variable. If you specify *ierr* on the call to CHECKMS or CHECKDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file. Otherwise, an error code is returned, and the message is added to the program's stderr file. On output from CHECKMS or CHECKDR, *ierr* contains one of the following error codes:

= 0 No error detected.

< 0 Error detected. *ierr* contains one of the following error codes:

−1 The file name or unit number is not valid.

−15 OPENMS/OPENDR was not called on this file.

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking, and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

ASYNCMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), READMS(3F), STINDX(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F)

NAME

CLOSEV – Closes volume and mounts next volume in Volume Identifier list

SYNOPSIS

CALL CLOSEV(*unit*, *istat*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CLOSEV closes a volume and mounts the next volume.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

istat Result of CLOSEV:
- =0 No error occurred
- ≠0 Error or warning occurred

NOTES

The file must be an opened tape file.

SEE ALSO

CHECKTP(3F), ENDSP(3F), SETSP(3F), STARTSP(3F)

NAME

CLOSMS, CLOSDR – Writes master index and closes random-access file

SYNOPSIS

```
CALL  CLOSMS(dn [,ierr])
CALL  CLOSDR(dn [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

CLOSMS/CLOSDR writes the master index specified in OPENMS/OPENDR from the user program area to the random-access file and then closes the file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

ierr Error control and code. Specify an integer variable. If you specify *ierr* on the call to CLOSMS/CLOSDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned and the message is added to the program's stderr file. On output from CLOSMS/CLOSDR, *ierr* has one of the following values:

= 0 No error detected.

< 0 Error detected. *ierr* contains one of the following error codes:

−1 The file name or unit number is not legal.

−15 OPENMS/OPENDR was not called on this file.

Statistics on the activity of the random-access file are written to the stderr file (see the following table). CLOSMS/CLOSDR writes a message to stderr on closing the file.

Message	Description
`TOTAL ACCESSES =`	*number_of_accesses*
`READS =`	*number_of_reads*
`WRITES =`	*number_of_writes*
`SEQUENTIAL READS =`	*number_of_sequential reads*
`SEQUENTIAL WRITES =`	*number_of_sequential writes*
`REWRITES IN PLACE =`	*number_of_rewrites in place*
`WRITES TO EOI =`	*number_of_writes to EOI*
`TOTAL WORDS MOVED =`	*number_of_words moved*
`MINIMUM RECORD =`	*minimum_record_size*
`MAXIMUM RECORD =`	*maximum_record_size*
`TOTAL ACCESS TIME =`	*total_access_time*
`AVERAGE ACCESS TIME =`	*average_access_time*

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking, and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

CAUTIONS

If a program terminates without closing the random-access file with `CLOSMS/CLOSDR`, file integrity is questionable.

SEE ALSO

`OPENMS`(3F)

NAME

FINDMS – Reads record into data buffers used by random-access routines

SYNOPSIS

CALL FINDMS(*dn, n, irec* [,*ierr*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

FINDMS asynchronously reads the desired record into the data buffers used by the random-access file routines for the specified file. The next READMS(3F) or WRITMS(3F) call waits for the read to complete and transfers data appropriately.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

n The number of words to be read, as in READMS or WRITMS. Type integer variable, expression, or constant.

irec As in READMS or WRITMS, the record name or number to be read into the data buffers. Specify an integer variable, expression, or constant.

ierr Error control and code. Specify a type integer variable. If you specify *ierr* on the call to FINDMS, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put in the stderr file; otherwise, an error code is returned, and the message is added to the stderr file.

On output from FINDMS, *ierr* has one of the following values:

= 0 No errors or warnings detected.

< 0 Error or warning condition detected. *ierr* contains one of the following error or warning codes:

- −6 The user-supplied named index is not valid.
- −8 The index number is greater than the maximum on the file.
- −10 The named record was not found in the index array.
- −15 OPENMS/OPENDR was not called on this file.
- −17 The index entry is less than or equal to 0 in the user's index array.
- −18 The user-supplied word count is less than or equal to 0.
- −19 The user-supplied index number is less than or equal to 0.
- −22 The record does not fit in the buffer. (warning)

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or if microtasking and you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), OPENMS(3F), READMS(3F), STINDX(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F),

NAME

FLUSH – Writes data buffered by Fortran output statements to a file

SYNOPSIS

Cray PVP systems and Cray MPP systems:
CALL FLUSH(*iunit*)

All Cray Research systems and SPARC systems:
CALL FLUSH(*iunit*),[*istat*]

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

FLUSH writes to a file any buffered data previously written by Fortran output statements. FLUSH may be called after reading or writing. The current file position is not changed.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

FLUSH has the following arguments:

iunit Integer variable, expression, or constant containing a Fortran unit number. If this value is 101, it flushes the unit which is connected to standard output.

istat Integer variable or array element that receives the return status as follows:

- =0 Indicates that all buffered data, if any, was flushed. A file that was immediately read, or a file with a buffer that has not changed since the last flush, might return a 0 status even though no new data would be written to the file.
- −1 Indicates that the Fortran unit does not support FLUSH, and no data was written to the file.
- >0 Indicates that an error condition was encountered. The specific error number is returned. Use the explain(1) command for a description of the error code.

The *istat* argument is optional; the user program is aborted if FLUSH is called without *istat* and an error condition is encountered.

NOTES

If FLUSH is called with an unconnected unit, an error condition results. Calling FLUSH for a file for which the program does not have write permission returns a 0 status if no data was written to that file previously with Fortran I/O statements.

In the UNICOS 6.1 and 7.0 releases, FLUSH is supported for sequential formatted I/O on all Cray Research systems. If the assign(1) option -s is specified for the file, the option value must be text or sbin.

A WRITE I/O statement with an asterisk unit identifier (or a PRINT statement) uses unit 101. See INTRO_IO(3F) for more information.

SEE ALSO

explain(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011, to see a description of the error code

NAME

FSUP, ISUP – Suppress values in Fortran edit-directed output

SYNOPSIS

```
CALL FSUP(fvalue)
CALL ISUP(ivalue)
CALL FSUPC
CALL ISUPC
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

fvalue and *ivalue* are real and integer arguments of default KIND. When FSUP is called, all REAL values equal to *fvalue* are suppressed (are output as blanks) when they are encountered in a Fortran edit-directed formatted I/O operation. FSUP may be recalled to redefine itself.

FSUPC undoes the call to FSUP, and all types are output as ordinary Fortran I/O.

FSUPC and ISUPC invalidate the function obtained by calling FSUP or ISUP, returning to ordinary Fortran I/O.

ISUP and ISUPC are the integer equivalents of FSUP and FSUPC.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

NAME

GETTP – Gets information about an opened tape file

SYNOPSIS

CALL GETTP(*unit*, *len*, *ipa*, *synch*, *istat*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GETTP retrieves information about an opened tape file, including its current status.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

len Integer length (in Cray words) of information array *ipa*. GETTP uses this parameter to determine the maximum number of information words to return. This parameter also allows for the addition of more fields in an upward-compatible manner. Currently, 48 words are defined.

ipa Integer information array. On exit, this array contains tape information, as follows (see the NOTES section for details about use on Cray MPP systems):

- *ipa*(1) Current volume identifier (left-justified, zero-filled).
- *ipa*(2)–*ipa*(7) Characters 1 through 48 of the path name of the file opened to this tape (left-justified, zero-filled).
- *ipa*(8) Integer file section number.
- *ipa*(9) Integer file sequence number.
- *ipa*(10) Integer block number relative to tape mark number specified in *ipa*(23).

ipa(11) Integer number of blocks in the tape layer library buffer. If additional processing layers have been specified with `assign`(1) or `asgcmd`(1), those layers may also hold buffered data, but they will not be included in this field.

ipa(12) Integer number of blocks in the IOP buffer.

ipa(13) Device ID or unit number in integers.

ipa(14) Device identifier or name (left-justified, zero-filled).

ipa(15) Generic device name (left-justified, zero-filled).

ipa(16) Last device function as an integer.

ipa(17) Last device status as an integer.

ipa(18) Data transfer count in bytes as an integer.

ipa(19) Buffer memory sector count as an integer.

ipa(20) Partial block bytes in buffer memory as an integer.

ipa(21) Outstanding sector count as an integer.

ipa(22) Outstanding block count as an integer.

ipa(23) User tape mark number as an integer, including tape marks embedded in the data.

ipa(24) Direction from tape mark in previous word: 0= after tape mark and 1= before tape mark.

ipa(25) Today's year modulus 100 as an integer.

ipa(26) Today's Julian day as an integer.

ipa(27) File identifier, up to the first 8 characters (left-justified, zero-filled).

ipa(28) Record format name (left-justified, zero-filled).

ipa(29) Tape density as an integer: 1=1600 b/i, 2= 6250 b/i.

ipa(30) Maximum block size as an integer.

ipa(31) Record length as an integer.

ipa(32) File status as an integer: 1= new, 2= old, 3= append.

ipa(33) Label type as an integer: 1= no label, 2= ANSI label, 3= IBM standard label, 4= bypass label.

ipa(34) Integer file sequence number of first file on volume.

ipa(35) Ring status as an integer: 0= ring out, 1= ring in.

ipa(36) Expiration year modulus 100 as an integer.

ipa(37) Expiration Julian day as an integer.

ipa(38) First volume identifier of file as an integer.

ipa(39) User end-of-volume processing status as an integer: 0= EOV processing off,
1= EOV processing on.

ipa(40) User end-of-volume processing status as an integer: 0= EOV processing is not currently active, 1= EOV processing is currently active.

ipa(41) User read/write tape mark status as an integer: 0= user read/write tape mark is not allowed, 1= user read/write tape mark allowed.

ipa(42) Block attribute (left-justified, zero-filled).

'B'L=	blocked records
'S'L=	spanned records if the record format is 'V', or standard records if the record format is 'F'
'R'L=	blocked and spanned records if the record format is 'V'
'R'L=	blocked and standard records if the record format is 'F'
' 'L (blank)=	none of the above

ipa(43)–*ipa*(48)
File identifier (left-justified, zero-filled).

synch This parameter is ignored if the last I/O operation was a read. If the last operation was a write, this parameter is meaningful. In this case, the following values apply:

`=0` Do not synchronize dataset.

`=1` Synchronize dataset before obtaining position information. It is invalid to specify this value in the following circumstance: end-of-volume (EOV) processing is enabled, and the user has reached the EOV but has not started special processing. This parameter has no effect on data buffered in library layers other than the tape layer.

istat Integer return status. On exit, this parameter contains a value indicating the status of the information request:

`=0` Indicates that the file was successfully queried, and *ipa* array is defined.

`≠0` Indicates that an error was encountered during an attempt to query the tape file, or an invalid argument was passed to `GETTP` (for example, *len*≤0). The *ipa* array is undefined.

NOTES

On Cray MPP systems, `GETTP` is supported for IBM-compatible tapes with the `er90` FFIO layer. This support may be removed in a future release. The `er90` layer does not guarantee that records correspond to physical tape blocks. See the `assign`(1) man page and the *UNICOS Tape Subsystem User's Guide*, publication SG–2051, for more information about the `er90` FFIO layer.

SEE ALSO

`SETTP`(3F)

NAME

GETWA, SEEK – Synchronously and asynchronously reads data from the word-addressable, random-access file

SYNOPSIS

CALL GETWA(*dn, result, addr, count* [,*ierr*])

CALL SEEK(*dn, addr, count* [,*ierr*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Use the SEEK and GETWA calls together when possible to improve efficiency; SEEK calls are never functionally required. The SEEK call reads the data asynchronously; the GETWA call waits for I/O to complete and then transfers the data. The SEEK call moves the last write operation pages from memory to disk, loading the user-requested word addresses to the front of the I/O buffers. You can load as much data as fits into the file buffers. Subsequent GETWA and PUTWA calls that reference word addresses in the same range do not cause any disk I/O.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

result Variable or array of any type. The location in the user program at which the first word is placed.

addr For GETWA, the word location of the file from which the first word is transferred. For SEEK, the word address of the next read. Specify a type integer variable, expression, or constant.

count For GETWA, the number of words written from the file into user memory. For SEEK, the number of words of the next read. Specify a type integer variable, expression, or constant.

ierr Error control and code. Specify a type integer variable. If you specify *ierr* on the call to GETWA or SEEK, *ierr* returns any error codes to you. If *ierr* is not specified, an error aborts the program.

On output from GETWA, *ierr* has one of the following values:

= 0 No errors detected.

<0 Error detected. *ierr* contains one of the following error codes:

+1 Request for a seek is outside page buffers.

−1 Invalid unit number.

−2 The number of files has exceeded memory or size availability.

−3 User attempt to read past end-of-data (EOD).

−4 User-supplied word address less than or equal to 0.

−5 User-requested word count greater than maximum allowed.

−6 Illegal file name.

−7 User word count less than or equal to 0.

NOTES

Most of the routines in the run-time libraries are reentrant or have internal locks to ensure that they are single threaded. Some library routines, however, must be locked at the user level if they are used by more than one task.

GETWA is not internally locked. You must lock each call to GETWA if it is called from more than one task.

You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to the stderr file.

It is not valid to mix calls to word-addressable routines with standard Fortran I/O statements that use the same unit number.

EXAMPLES

Assume that you want to use a routine that reads word addresses 1,000,000 to 1,051,200. A file is opened with 101 blocks of buffer space, and CALL SEEK(*filename*,1000000,51200,*ierr*) is used before calling the routine. Subsequent GETWA or PUTWA calls with word addresses in the range of 1,000,000 to 1,051,200 do not trigger any disk I/O.

SEE ALSO

PUTWA(3F), WCLOSE(3F), WOPEN(3F)

NAME

GTSTDPTR – Returns pointer to standard file

SYNOPSIS

```
INTEGER fd, stream, GTSTDPTR
stream = GTSTDPTR(fd)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GTSTDPTR maps a standard file descriptor (0, 1, or 2) into the corresponding address of the FILE structure (stream pointer) for subsequent use in the Fortran-callable fread(3C) and fwrite(3C) routines.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine has the following arguments:

fd Integer file descriptor. The following values are available:

- 0 Return address of standard input (stdin)
- 1 Return address of standard output (stdout)
- 2 Return address of standard error (stderr)

stream
Address of the FILE structure (stream pointer). Null (0) if the file descriptor is not valid (that is, if it is not 0, 1, or 2).

SEE ALSO

fread(3C), fwrite(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

NUMBLKS – Returns the current size of a file in 512-word blocks

SYNOPSIS

```
INTEGER  NUMBLKS
val  =  NUMBLKS (unit)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

NUMBLKS returns the current size of a file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

val File size in 512-word blocks. This returned value reflects only the data actually written to disk and does not take into account data still in the buffers. A value of 0 is returned if the file is a pipe or character special device. A negative value indicates that *unit* is not connected or a system operation failed unexpectedly.

unit An integer variable, expression, or constant containing a Fortran unit number.

NAME

OPENMS, OPENDR – Opens a local file as a random-access file that can be accessed or changed by the record-addressable, random-access file I/O routines

SYNOPSIS

```
CALL OPENMS(dn, index, length, it [,ierr])
CALL OPENDR(dn, index, length, it [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

OPENMS/OPENDR opens a file and specifies it as a random-access file that can be modified by the random-access I/O routines. If the file does not exist, the master index contains zeros; if it does exist, the master index is read from the file. The master index contains the current index to the file that is updated when the file is closed by using CLOSMS/CLOSDR.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The default number of open files is 20 for OPENDR and 40 for OPENMS.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

index The name of the array in the user program that is going to contain the master index to the records of the file. Specify a type integer array. This array must be changed only by the random-access file I/O routines. *index* should be a multiple of 512 words.

length The length of the index array. Specify a type integer variable, expression, or constant. The length of *index* depends on the number of records on, or to be written to, the file, using the master index and on the type of master index. The *length* specification must be at least 2**nrec* if *it*=1 or 3, or *nrec* if *it*=0 or 2. *nrec* is the number of records in, or to be written to, the file, using the master index.

it Flag indicating the type of master index. Specify a type integer variable, expression, or constant. *it* can have one of the following values:

- `0` Records synchronously referenced with a number between 1 and *length*
- `1` Records synchronously referenced with an alphanumeric name of 8 or fewer characters
- `2` Records asynchronously referenced with a number between 1 and *length*
- `3` Records asynchronously referenced with an alphanumeric name of 8 or fewer characters

For a named index, odd-numbered elements of the index array contain the record name, and even-numbered elements of the index array contain the pointers to the location of the record within the file. For a numbered index, a given index array element contains the pointers to the location of the corresponding record within the file.

ierr Error control and code. Specify a type integer variable. If you specify *ierr* on the call to `OPENMS/OPENDR`, *ierr* returns any error codes to you. If *ierr* is not specified, an error aborts the program. If you set *ierr*`>0` on input to `OPENMS/OPENDR`, error messages are not placed in the `stderr` file; otherwise, an error code is returned, and the error message is added to the `stderr` file. `OPENMS/OPENDR` writes an open message to the `stderr` file whether or not the value of *ierr* selects messages. On output from `OPENMS/OPENDR` *ierr* has the following values:

- `= 0` No errors detected.
- `< 0` Error detected. *ierr* contains one of the following error codes:
  - `−1` The file name or unit number is not valid.
  - `−2` The user-supplied index length is less than or equal to 0.
  - `−3` The number of files has exceeded memory or size availability.
  - `−4` The file index length read from the file is greater than the user-supplied index length (nonfatal message).
  - `−5` The user-supplied index length is greater than the index length read from the file (nonfatal message).

−11 The index word address read from the file is less than or equal to 0.

−12 The index length read from the file is less than 0.

−13 The file has a checksum error.

−14 OPENMS has already opened the file.

−20 File created by WRITDR/WRITMS.

−21 Memory limit exceeded while allocating internal tables.

NOTES

A file opened with OPENMS should be closed only by CLOSMS(3F). If you close the file in some other way, the future behavior of the program is unpredictable.

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking, and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), FINDMS(3F), READMS(3F), STINDX(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F)

NAME

PUTWA, APUTWA – Writes to a word-addressable, random-access file

SYNOPSIS

```
CALL PUTWA(dn, source, addr, count [,ierr])
CALL APUTWA(dn, source, addr, count [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

PUTWA synchronously writes a number of words from memory to a word-addressable, random-access file. APUTWA asynchronously writes a number of words from memory to a word-addressable, random-access file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

source Variable or array of any type. The location of the first word in the user program to be written to the file.

addr The word location of the file that is to receive the first word from the user program. *addr*=1 indicates beginning of file. Specify a type integer variable, expression, or constant.

count The number of words from source to be written. Specify a type integer variable, expression, or constant.

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to PUTWA/APUTWA, *ierr* returns any error codes to you. If *ierr* is not supplied, an error causes the program to abort. On return from PUTWA/APUTWA, *ierr* has one of the following values:

0	No errors detected.
+1	Request did not fit in the page table (buffer cache) in one pass; therefore, request was not truly asynchronous.
−1	Invalid unit number.
−2	Number of files has exceeded memory size availability.
−4	User-supplied word address less than or equal to 0.
−5	User-requested word count greater than maximum allowed.
−6	Invalid file name.
−7	User word count less than or equal to 0.

NOTES

Most of the routines in the run-time libraries are reentrant or have internal locks to ensure that they are single-threaded. Some library routines, however, must be locked at the user level if they are used by more than one task.

PUTWA is not internally locked. You must lock each call to PUTWA if it is called from more than one task. You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to stderr.

It is not valid to mix calls to word-addressable routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

GETWA(3F), WCLOSE(3F), WOPEN(3F)

NAME

READ, READP – Reads words, full or partial record modes

SYNOPSIS

CALL READ(*unit*, *word*, *count*, *status*, [*ubc*])

CALL READP(*unit*, *word*, *count*, *status*, [*ubc*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

READ and READP routines move words of data from disk or tape to a variable or array. After reading less than a full record from disk or tape, READ leaves the file positioned at the beginning of the next record, while READP leaves the file positioned at the next item in the record just read.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable, expression, or constant. *unit* can be a unit number or name of the file as a Hollerith constant of 7 characters or less ('MYFILE'L).

word Word-receiving data area, such as a variable or array.

count Integer variable. On entry, the number of words requested. (Do not specify a constant.) On exit, the number of words actually transferred.

status Integer variable. On exit, *status* has one of the following values:

=−1 Words remain in record

= 0 End-of-record (EOR)

= 1 Null record

= 2 End-of-file (EOF)

= 3 End-of-data (EOD)

= 4 Hardware error

≥ 5 Error. The value may represent a Fortran library error or a system error. If it is a system error, a definition may be found in <errno.h>. See the explain(1) command for further details. See intro(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012, for a description of the system error names and numbers.

ubc Integer variable. Optional unused bit count. Number of unused bits contained in the last word of the record.

NOTES

If foreign record translation is enabled for the specified unit, the bits from the foreign logical records are delivered to the user data area without data conversion.

EXAMPLES

The following example reads the first 2 words of the next record on unit 15. It also checks for success by examining the returned value of *status* and writing a message containing the number of words read.

```
INTEGER REC(10)
IUNIT = 15
NUM = 2
CALL READ(IUNIT, REC, NUM, ISTAT)
IF (ISTAT .GT. 1) THEN
C       handle exceptional conditions (EOF, EOD, error)
  CALL ABORT('unexpected status encountered')
ENDIF
WRITE(*,*)' Read ' ,NUM,' actual words'
END
```

SEE ALSO

ACPTBAD(3F), READC(3F), READIBM(3F), SKIPBAD(3F), WRITE(3F), WRITEC(3F), WRITIBM(3F)

NAME

READC, READCP – Reads characters, full or partial record mode

SYNOPSIS

```
CALL READC(unit, char, count, status)
CALL READCP(unit, char, count, status)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

READC and READCP routines unpack characters from the I/O buffer and insert them in the user data area beginning at the first word address. Characters are placed into the data area, 1 character per word, right-justified. This process continues until the count is satisfied or an end-of-record (EOR) is encountered. If an EOR is encountered first, the remainder of the field specified by the character count is filled with blanks.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable, expression, or constant. *unit* can be a unit number or file name as a Hollerith constant of 7 characters or less ('MYFILE'L).

char Character-receiving data area.

count Integer variable. On entry, the number of characters requested. On exit, the number of characters actually transferred.

status Integer variable. On exit, *status* has one of the following values:

=−1 Characters remain in record

= 0 End-of-record (EOR)

= 1 Null record

= 2 End-of-file (EOF)

= 3 End-of-data (EOD)

= 4 Hardware error

≥ 5 Error. If it is a system error, a definition may be found in <errno.h>. See explain(1) for complete details. See intro(2) in the *UNICOS System Calls Reference Manual*, publication SR−2012, for a description of the system error names and numbers.

NOTES

If foreign record translation is enabled for the specified unit, the bits from the foreign logical records are delivered after character conversion as specified with the assign(1) command.

SEE ALSO

ACPTBAD(3F), READ(3F), READIBM(3F), SKIPBAD(3F), WRITE(3F), WRITEC(3F), WRITIBM(3F)

explain(1) in *UNICOS User Commands Reference Manual*, publication SR−2011

NAME

READMS, READDR – Reads a record from a random-access file to memory

SYNOPSIS

```
CALL READMS(dn, ubuff, n, irec [,ierr])
CALL READDR(dn, ubuff, n, irec [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

READMS and READDR read records from a random-access file to a contiguous memory area in the user's program.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

ubuff The location in your program where the first word of the record is placed. User-specified type; do not use type character.

n The number of words to be read. Specify a type integer variable, expression, or constant. *n* words are read from the random-access record *irec* and placed contiguously in memory, beginning at *ubuff*. If necessary, READDR rounds *n* up to the next multiple of 512 words. If the file is in synchronous mode, the data is saved and restored after the read. The maximum record size is limited to 4,194,303 words.

irec The record number or record name of the record to be read. Specify a type integer variable, expression, or constant. A record name is limited to a maximum of 8 characters. For a numbered index, *irec* must be between 1 and the length of the index declared in the OPENMS/OPENDR(3) call, inclusive. For a named index, *irec* is any 64-bit entity you specify.

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to READMS/READDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned, and the message is added to the stderr file.

On output from READMS/READDR, *ierr* has one of the following values:

= 0 No errors detected.

< 0 Error detected. *ierr* contains one of the following error codes:

- −1 The file name or unit number is invalid.
- −6 The user-supplied named index is invalid.
- −8 The index number is greater than the maximum on the file.
- −10 The named record was not found in the index array.
- −15 OPENMS/OPENDR was not called on this file.
- −17 The index entry is less than or equal to 0 in the user's index array.
- −18 The user-supplied word count is less than or equal to 0.
- −19 The user-supplied index number is less than or equal to 0.

NOTES

Most of the routines in the run-time libraries are reentrant or have internal locks to ensure that they are single threaded. Some library routines, however, must be locked at the user level if they are used by more than one task.

READMS and READDR are not internally locked. You must lock each call to these routines if they are called from more than one task.

You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to stderr.

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking and if you are multitasking your I/O calls.

It is not valid to mix calls to word-addressable routines with standard Fortran I/O statements that use the same unit number.

CAUTIONS

If you are using READDR in asynchronous mode, and the record size is not a multiple of 512 words, user data can be overwritten and not restored. With SYNCDR(3F), the file can be switched to read synchronously, causing data to be copied out and restored after the read has completed.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), STINDX(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F)

NAME

STINDX, STINDR – Allows an index to be used as the current index by creating a subindex

SYNOPSIS

```
CALL STINDX(filename, index, length, it [,ierr])
CALL STINDR(filename, index, length, it [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

STINDX/STINDR reduces the amount of memory needed by a file containing a large number of records. It also maintains a file containing records logically related to each other. Records in the file, rather than records in the master index area, hold secondary pointers to records in the file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

filename An integer variable, expression, or constant that has a maximum of 8 characters. The name of the file as a Hollerith constant. Alternatively, *filename* can be a positive integer that identifies the file as fort.*n*, where *n* = *filename*. If an integer value is passed in *filename*, it must be in the range 0 < *n* < 100.

index The user-supplied array used for the subindex or new current index. Specify a type integer array. If *index* is a subindex, it must be a storage area that does not overlap the area used in OPENMS/OPENDR(3) to store the master index.

length The length of the index array. Specify a type integer variable, expression, or constant. The length of *index* depends on the number of records in the file or to be written to the file using the master index and on the type of master index. If *it*=1, *length* must be at least twice the number of records in the file or to be written to the file using *index*. If *it*=0, *length* must be at least the number of records on or to be written to the file using *index*.

it A flag to indicate the type of index. Specify a type integer variable, expression, or constant. When *it*=0, the records are referenced with a number between 1 and *length*. When *it*=1, the records are referenced with an alphanumeric name of 8 or fewer characters. For a named index, odd-numbered elements of the index array contain the record name and even-numbered elements of the index array contain pointers to the location of the record within the file. For a numbered index, a given index array element contains pointers to the location of the corresponding record within the file. The index type defined by STINDX/STINDR must be the same as that used by OPENMS/OPENDR.

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to STINDX/STINDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned, and the message is added to the stderr file.

On return from STINDX/STINDR, *ierr* has one of the following values:

= 0 No errors detected.

< 0 Error detected. *ierr* contains one of the error codes described in the following table:

−1 The file name or unit number is not valid.

−15 OPENMS/OPENDR was not called on this file.

−16 A STINDX/STINDR call cannot change the index type.

STINDX/STINDR allows more than one index to manipulate the file. Generally, STINDX/STINDR toggle the index between the master index (maintained by OPENMS/OPENDR and CLOSMS/CLOSDR(3)) and a subindex (supplied and maintained by you).

You must maintain and update subindex records stored in the file. You can access and change records in the file only by using the current index.

After a STINDX/STINDR call, subsequent calls to READMS/READDR(3) and WRITMS/WRITDR(3) use and alter the current index array specified in the STINDX/STINDR call. You can save the subindex by calling STINDX/STINDR with the master index array, then writing the subindex array to the file using WRITMS/WRITDR. Retrieve the subindex by calling READMS/READDR on the record containing the subindex information. Thus, STINDX/STINDR allows logically infinite index trees into the file and reduces the amount of memory needed for a random-access file containing many records.

CAUTIONS

When generating a new subindex (for example, building a database), set the array or memory area used for the subindex to 0. If the subindex storage is not set to 0, unpredictable results occur.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), READMS(3F), SYNCMS(3F), WAITMS(3F), WRITMS(3F)

NAME

SYNCMS, SYNCDR – Sets I/O mode for random-access routines to synchronous

SYNOPSIS

```
CALL SYNCMS(dn [,ierr])
CALL SYNCDR(dn [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

SYNCMS and SYNCDR set I/O mode for random-access routines.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

ierr Error control and code. Specifies an integer variable. If you supply *ierr* on the call to SYNCMS/SYNCDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned, and the message is added to the stderr file.

On return from SYNCMS/SYNCDR, *ierr* has one of the following values:

= 0 No errors detected.

< 0 Error detected. *ierr* contains one of the following error codes:

−1 The file name or unit number is invalid.

−15 OPENMS/OPENDR was not called on this file.

All I/O operations wait for completion.

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), READMS(3F), STINDX(3F), WAITMS(3F), WRITMS(3F)

NAME

WAITMS, WAITDR – Waits for completion of an asynchronous I/O operation

SYNOPSIS

CALL WAITMS(*dn*, *istat* [,*ierr*])

CALL WAITDR(*dn*, *istat* [,*ierr*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WAITMS and WAITDR wait for the completion of an asynchronous I/O operation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn Integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

istat File Error flag. Specify a type integer variable; *istat* can have one of the following values:

- 0 No error occurred during the asynchronous I/O operation
- 1 Error occurred during the asynchronous I/O operation

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to WAITMS/WAITDR, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the stderr file; otherwise, an error code is returned, and the message is added to the stderr file.

On return from WAITMS/WAITDR, *ierr* has one of the following values:

- =0 No errors detected.
- <0 Error detected. *ierr* contains one of the following error codes:
  - −1 The file name or unit number if not valid.
  - −15 OPENMS/OPENDR was not called on this file.

NOTES

The random-access I/O routines cannot be used in multitasking unless you provide your own locks. If you are Autotasking or if you are not multitasking at all, you will not be affected. You will need locks only if you are doing your own macrotasking or microtasking and if you are multitasking your I/O calls.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

ASYNCMS(3F), CHECKMS(3F), CLOSMS(3F), FINDMS(3F), OPENMS(3F), READMS(3F), SYNCMS(3F), STINDX(3F) WRITMS(3F)

NAME

WCLOSE – Finalizes changes and closes word-addressable, random-access file

SYNOPSIS

CALL WCLOSE(*dn* [,*ierr*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WCLOSE finalizes the additions and changes to the word-addressable, random-access file and closes the file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

ierr Error control and code. Specify an integer variable, expression, or constant. If you supply *ierr* on the call to WCLOSE, *ierr* returns any error codes to you. If *ierr* is not supplied, an error aborts the program.

On output from WCLOSE, *ierr* has one of the following values:

0	No errors detected
−1	Invalid unit number
−6	Invalid file name

NOTES

You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to stderr.

It is not valid to mix calls to word-addressable routines with standard Fortran I/O statements that use the same unit number.

SEE ALSO

GETWA(3F), PUTWA(3F), WOPEN(3F)

NAME

WOPEN – Opens a word-addressable, random-access file

SYNOPSIS

CALL WOPEN(*dn*, *blocks*, *istats* [,*ierr*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WOPEN opens a file and specifies it as a word-addressable, random-access file that can be accessed or changed with the word-addressable I/O routines. The WOPEN call is optional.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn An integer variable, expression, or constant with a maximum of 8 characters. *dn* can be a unit number or the name of the file as a Hollerith constant.

blocks The maximum number of 512-word blocks that the word-addressable package can use for a buffer. Specify a type integer variable, expression, or constant.

istats Specify a type integer variable, expression, or constant. If *istats* is nonzero, statistics about the changes and accesses to the file *filename* are collected. (See the following table for information about the statistics that are collected.) Statistics are written to stderr.

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to WOPEN, *ierr* returns any error codes to you. If *ierr* is not supplied, an error aborts the job.

On output from WOPEN, *ierr* has one of the following values:

0	No errors detected
−1	Invalid unit number
−2	Number of files has exceeded memory size availability
−6	Invalid file name

NOTES

A file opened using WOPEN should be closed only by WCLOSE(3F). If you close the file in some other way, the subsequent behavior of the program and the validity of the file are unpredictable.

If you bypass WCLOSE, the internal tables maintained by the word-addressable I/O package are not updated, leaving dangling pointers in future computation.

You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to stderr.

It is not valid to mix calls to word-addressable routines with standard Fortran I/O statements that use the same unit number.

MESSAGES

Message	Description
BUFFERS USED =	Number of 512-word buffers used by this file.
TOTAL ACCESSES =	Number of accesses. This is the sum of the GETWA(3F) and PUTWA(3F) calls.
GETS =	Number of times the user called GETWA.
PUTS =	Number of times the user called PUTWA.
FINDS =	Number of times the user called SEEK(3F).
HITS =	Number of times word addresses desired were resident in memory.
MISSES =	Number of times no word addresses desired were resident in memory.
PARTIAL HITS =	Number of times that some but not all of the word addresses desired were in memory.
DISK READS =	Number of physical disk reads done.
DISK WRITES =	Number of times a physical disk was written to.
BUFFER FLUSHES =	Number of times buffers were flushed.
WORDS READ =	Number of words moved from buffers to user.
WORDS WRITTEN =	Number of words moved from user to buffer.
TOTAL WORDS =	Sum of WORDS READ and WORDS WRITTEN.
TOTAL ACCESS TIME =	Real time spent in disk transfers.
AVER ACCESS TIME =	TOTAL ACCESS TIME divided by the sum of DISK READS and DISK WRITES.
EOD BLOCK NUMBER =	Number of the last block of the file.

Message	Description
DISK WORDS READ =	Count of number of words moved from disk to buffers.
DISK WDS WRITTEN =	Count of number of words moved from buffers to disk.
TOTAL DISK XFERS =	Sum of DISK WORDS READ and DISK WORDS WRITTEN.
BUFFER BONUS % =	TOTAL WORDS divided by value TOTAL DISK XFERS multiplied by 100.

SEE ALSO

GETWA(3F), PUTWA(3F), WCLOSE(3F)

NAME

WRITE, WRITEP – Writes words, full or partial record mode

SYNOPSIS

```
CALL WRITE(unit, word, count, ubc, status)
CALL WRITEP(unit, word, count, ubc, status)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WRITE and WRITEP write words in full or partial record mode.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of arguments for this routine.

unit An integer variable, expression, or constant. *unit* can be a unit number or file name as a Hollerith in 7 characters or less ('MYFILE'L).

word Data area containing words; this may not be a Fortran character variable.

count An integer variable. Word count. For WRITE, a value of 0 causes an end-of-record (EOR) record control word to be written.

ubc An integer variable. Optional unused bit count. Number of unused bits contained in the last word of the record.

status An integer variable. Optional status. If present, *status* has one of the following values on exit:

- =0 No error.
- ≠0 Error. If it is a system error, a definition may be found in <errno.h>. See the explain(1) command for complete details. See intro(2) in *UNICOS System Calls Reference Manual*, publication SR–2012, for a description of the system error names and numbers.

In routines in which words are written, the number of words specified by the count are transmitted from the area beginning at the first word address and are written in the I/O buffer.

NOTES

If foreign record translation is enabled for the specified unit, the bits from the user are delivered to the foreign logical records without data conversion.

SEE ALSO

ACPTBAD(3F), READ(3F), READC(3F), READIBM(3F), SKIPBAD(3F), WRITEC(3F), WRITIBM(3F)

explain(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

WRITEC, WRITECP – Writes characters, full or partial record mode

SYNOPSIS

```
CALL WRITEC(unit, char, count, status)
CALL WRITECP(unit, char, count, status)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WRITEC and WRITECP write characters in full or partial record mode.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable, expression, or constant. *unit* can be a unit number or file name as a Hollerith constant in 7 characters or less ('MYFILE'L).

char Data area containing characters; this may not be a Fortran character variable.

count Type integer variable. Character count.

status Type integer variable. Optional status. If present, *status* has one of the following values on exit:

- =0 No error.
- ≠0 Error. If it is a system error, a definition may be found in <errno.h>. See intro(2) in *UNICOS System Calls Reference Manual*, publication SR–2012, for a description of the system error names and numbers.

WRITEC and WRITECP routines pack characters into the I/O buffer for the file. The count specifies the number of characters packed. These characters originate from the user area defined at the first word address, which is 1 character per source word (right-justified).

NOTES

If foreign record translation is enabled for the specified unit, the bits from the user are delivered to the foreign logical records after character conversion is specified with the `assign`(1) command.

SEE ALSO

`ACPTBAD`(3F), `READ`(3F), `READC`(3F), `READIBM`(3F), `SKIPBAD`(3F), `WRITE`(3F), `WRITIBM`(3F)

`assign`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

WRITMS, WRITDR – Writes to a random-access file on disk

SYNOPSIS

```
CALL WRITMS(dn, ubuff, n, irec, rrflag, s [,ierr])
CALL WRITDR(dn, ubuff, n, irec, rrflag, s [,ierr])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

WRITMS and WRITDR write data from user memory to a record in a random-access file on disk and update the current index.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dn Type integer variable, expression, or constant with a maximum of 8 characters. *dn* is a unit number or the name of the file as a Hollerith constant.

ubuff The location of the first word in the user program to be written to the record. User-specified type. May not be a Fortran character variable.

n The number of words to be written to the record. Specify a type integer variable, expression, or constant. *n* contiguous words from memory, beginning at *ubuff*, are written to the file record. Because UNICOS unblocked-file I/O is in multiples of 512 words, it is recommended that *n* be a multiple of 512 words when speed is important. However, the random access file I/O routines support record lengths other than multiples of 512 words. WRITDR rounds *n* up to the next multiple of 512 words, if necessary. The maximum record size is limited to 4,194,303 words.

irec The record number or record name of the record to be written. Specify a type integer variable, expression, or constant. A record name is limited to a maximum of 8 characters. For a numbered index, *irec* must be between 1 and the length of the index declared in the OPENMS/OPENDR(3) call. For a named index, *irec* is any 64-bit entity you specify.

rrflag A flag indicating record rewrite control. Specify a type integer variable, expression, or constant. *rrflag* can be one of the following codes:

- `0` Write the record at end-of-data (EOD).
- `1` If the record already exists, and the new record length is less than or equal to the old record length, rewrite the record over the old record. If the new record length is greater than the old, abort the program or return the error code in *ierr*. If the record does not exist, the program aborts or the error code is returned in *ierr*.
- `-1` If the record exists, and its new length does not exceed the old length, write the record over the old record; otherwise, write the record at EOD.

s A subindex flag. Specify a type integer variable, expression, or constant. (The implementation of this parameter has been deferred.)

ierr Error control and code. Specify a type integer variable. If you supply *ierr* on the call to `WRITMS/WRITDR`, *ierr* returns any error codes to you. If *ierr*>0, no error messages are put into the `stderr` file; otherwise, an error code is returned and the message is added to the `stderr` file.

On return from `WRITMS/WRITDR`, *ierr* has one of the following values:

- `0` No errors or warnings detected.
- `<0` Error or warning condition detected. *ierr* contains one of the following error or warning codes:

  Error codes:

  - `-1` The file name or unit number is not valid.
  - `-6` The user-supplied named index is not valid.
  - `-7` The named record index array is full.
  - `-8` The index number is greater than the maximum on the file.
  - `-9` Rewrite record exceeds the original.
  - `-15` `OPENMS/OPENDR` was not called on this file.
  - `-17` The index entry is less than or equal to 0 in the user's index array.
  - `-18` The user-supplied word count is less than or equal to 0.
  - `-19` The user-supplied index number is less than or equal to 0.

−26 The user-requested record length exceeds the maximum allowed.

Warning codes:

−21 The record size exceeds the buffer size for an asynchronous WRITMS request. The WRITMS operation operated synchronously.

NOTES

Most of the routines in the run-time libraries are reentrant or have internal locks to ensure that they are single threaded. Some library routines, however, must be locked at the user level if they are used by more than one task.

WRITMS and WRITDR are not internally locked. You must lock each call to these routines if they are called from more than one task.

You cannot use the word-addressable I/O package on tape. An attempt to use a tape file causes an error message to be written to stderr.

It is not valid to mix calls to MS/DR routines with standard Fortran I/O statements that use the same unit number.

EXAMPLES

The following examples show some of the features and uses of random-access file routines.

Example 1: In the SORT program, a sequence of records is read in and then printed out as a sorted sequence of records.

```
      PROGRAM SORT
      INTEGER IARRAY (512)
      INTEGER INDEX (512), KEYS (100)
      CALL OPENMS ('SORT'L,INDEX,255,1)
      N=50
C READ IN RANDOM ACCESS RECORDS FROM UNIT "SORT"
      DO 21 I=1,N
        READ(5,1000) (IARRAY(J),J=1,512)
        NAME=IARRAY(1)
        KEYS(I)=IARRAY(1)
       CALL WRITMS ('SORT'L,IARRAY,512,NAME,0)
21    CONTINUE
C SORT KEYS ALPHABETICALLY IN ASCENDING ORDER USING
C EXCHANGE SORT
      DO 23 I=1,N-1
        MIN=I
        J=I+1
        DO 22 K=J,N
          IF (KEYS(K).LT.KEYS(MIN)) MIN=K
22      CONTINUE
        IB=KEYS(I)
        KEYS(I)=KEYS(MIN)
        KEYS(MIN)=IB
23    CONTINUE
C WRITE OUT RANDOM ACCESS RECORDS IN ASCENDING
C ALPHABETICAL ORDER
      DO 24 I=1,N
        NAME=KEYS(I)
        CALL READMS ('SORT'L,IARRAY,512,NAME)
        WRITE(6,5120) (IARRAY(J),J=1,512)
24    CONTINUE
1000  FORMAT ("......")
5120  FORMAT (1X,"......")
      CALL CLOSMS ('SORT'L)
      STOP
      END
```

In this example, the random-access file is initialized as shown in line 4. Lines 6 through 11 show that a record is read from unit 5 into array IARRAY and then written as a record to the random-access file SORT. The first word of each record is assumed to contain an 8-character name to be used as the name of the record.

Lines 12 through 21 sort the names of the records in array KEYS. Lines 22 through 26 read in and then print the records in alphabetical order.

Example 2: Programs INITIAL and UPDATE show how the random-access file might be updated without the usual search and positioning of a sequential access file.

Program INITIAL:

```
      PROGRAM INITIAL
      INTEGER IARRAY(512)
      INTEGER INDEX (512)
C
C OPEN RANDOM ACCESS DATASET
C THIS INITIALIZES THE RECORD KEY "INDEX"
C
      CALL OPENMS (’MASTER’L,INDEX,101,1)
C
C READ IN RECORDS FROM UNIT 6 AND
C WRITE THEM TO THE DATASET "MASTER"
C
      DO 10 I=1,50
        READ(6,600) (IARRAY(J),J=1,512)
        NAME=IARRAY(1)
        CALL WRITMS (’MASTER’L,IARRAY,512,NAME,0,0)
     CONTINUE
C
C CLOSE "MASTER" AND SAVE RECORDS FOR UPDATING
C
10     CALL CLOSMS (’MASTER’L)
           600  FORMAT (1X,’.....’)
      STOP
      END
```

Program UPDATE:

```
      PROGRAM UPDATE
      INTEGER INEWRCD(512)
      INTEGER INDX (512)
C
C OPEN RANDOM ACCESS DATASET CREATED IN THE
C PREVIOUS PROGRAM "INITIAL"
C
C INDX WILL BE WRITTEN OVER THE OLD RECORD KEY
C
      CALL OPENMS (’MASTER’L,INDX,101,1)
C
C READ IN NUMBER OF RECORDS TO BE UPDATED
C
5      READ (6,610) N
C
C READ IN NEW RECORDS FROM UNIT 6 AND
C WRITE THEM IN PLACE OF THE OLD RECORD THAT HAS
C THAT NAME
C
      DO 10 I=1,N
        READ(6,600) (INEWRCD(J),J=1,512)
        NAME=INEWRCD(1)
        CALL WRITMS (’MASTER’L,INEWRCD,512,NAME,1,0)
10      CONTINUE
C
C CLOSE "MASTER" AND SAVE NEWLY UPDATED RECORDS
C FOR FURTHER UPDATING
C
      CALL CLOSMS (’MASTER’L)
 600  FORMAT (1X,"......")
 610  FORMAT (1X,"......")
      STOP
      END
```

In the preceding example, program INITIAL creates a random-access file on unit MASTER; program UPDATE then replaces particular records of this file without changing the remainder of the records.

Line 10 shows that the call to CLOSMS(3F) at the end of INITIAL caused the contents of INDEX to be written to the random-access file.

Line 4 shows that the call to OPENMS(3F) at the beginning of UPDATE has caused the record key of the random-access file to be written to INDX. The random-access file and INDX are now the same as the random-access file and INDEX at the end of INITIAL.

Lines 6 through 10 show that certain records are replaced.

Example 3: The program SNDYMS is an example of the use of the secondary index capability, using STINDX(3F). In this example, dummy information is written to the random-access file.

```
      PROGRAM SNDYMS
      IMPLICIT INTEGER (A-Y)
      DIMENSION PINDEX(20),SINDEX(30),ZBUFFR(50)
      DATA PLEN,SLEN,RLEN /20,30,50/
C OPEN THE FILE.
       CALL OPENMS (1,PINDEX,PLEN,0,ERR)
      IF (ERR.NE.0) THEN
         PRINT*,' Error on OPENMS, err=',ERR
         STOP 1
      ENDIF
C LOOP OVER THE 20 PRIMARY INDICES.  EACH TIME
C A SECONDARY INDEX IS FULL, WRITE THE
C SECONDARY INDEX ARRAY TO THE DATASET.
      DO 40 K=1,PLEN
C ZERO OUT THE SECONDARY INDEX ARRAY.
      DO 10 I=1,SLEN
10    SINDEX(I)=0
C CALL STINDX TO CHANGE INDEX TO SINDEX.
      CALL STINDX (1,SINDEX,SLEN,0,ERR)
      IF (ERR.NE.0) THEN
         PRINT*,' Error on STINDX, err=',ERR
         STOP 2
      ENDIF
C WRITE SLEN RECORDS.
      DO 30 J=1,SLEN
C GENERATE A RECORD LENGTH BETWEEN 1 AND RLEN.
      TRLEN=MAX0(IFIX(RANF(0)*FLOAT(RLEN)),1)
C FILL THE "DATA" ARRAY WITH RANDOM FLOATING POINT
C NUMBERS.
```

```
      DO 20 I=1,TRLEN
20    ZBUFFR(I)=(J+SIN(FLOAT(I)))**(1.+RANF(0))
      CALL WRITMS (1,ZBUFFR,TRLEN,J,-1,DUMMY,ERR)
      IF (ERR.NE.0) THEN
         PRINT*,' Error on WRITMS, err=',ERR
         STOP 3
      ENDIF
30    CONTINUE
C "TOGGLE" THE INDEX BACK TO THE MASTER AND
C WRITE THE SECONDARY INDEX TO THE DATASET.
      CALL STINDX (1,PINDEX,PLEN,0)
C NOTE THE ABOVE STINDX CALL DOES NOT USE THE
C OPTIONAL ERROR PARAMETER, AND WILL ABORT
C IF STINDX DETECTS AN ERROR.
      CALL WRITMS (1,SINDEX,SLEN,K,-1,DUMMY,ERR)
      IF (ERR.NE.0) THEN
         PRINT*,' Error on STINDX, err=',ERR
         STOP 4
      ENDIF
40 CONTINUE

C CLOSE THE DATASET.
      CALL CLOSMS (1,ERR)
      IF (ERR.NE.0) THEN
         PRINT*,' Error on CLOSMS, err=',ERR
         STOP 5
      ENDIF
      STOP 'Normal'
      END
```

SEE ALSO

CLOSMS(3F), OPENMS(3F), STINDX(3F)

NAME

ACPTBAD – Makes bad data available

SYNOPSIS

CALL ACPTBAD(*unit*, *uda*, *wrdcnt*, *termcnd*, *ubcnt* [,*acptcnt*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ACPTBAD makes bad data available to you by transferring it to the user-specified buffer.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit Unit number.

uda User data area to receive bad data.

wrdcnt On exit, number of words transferred.

termcnd On exit, address of termination condition:

- =0 Positioned at end-of-block
- =1 Positioned at end-of-file
- =2 Positioned at end-of-data
- <0 Not positioned at end-of-block

ubcnt On exit, unused bit count. Defined only if *termcnd* is 0, and *wrdcnt* is nonzero.

acptcnt Optional integer parameter. On input, it specifies the maximum number of words of bad data that you want to accept. If the number of words of bad data in the block exceeds this value, the excess is discarded.

EXAMPLES

Example 1 demonstrates the use of ACPTBAD when used with the READ library routine.

```
      PROGRAM EXAMPLE1
      IMPLICIT INTEGER(A-Z)
      PARAMETER(NBYTES=40000,NDIM=NBYTES/8,DN=99)
      DIMENSION BUFFER(1:NDIM)
      DIMENSION UDA(1:2*NDIM)


2000  CONTINUE
      NWORDS = NDIM
      CALL READ(DN,BUFFER,NWORDS,STATUS)
      IF((STATUS.EQ.2) .OR. (STATUS.EQ.3)) THEN    ! EOF or EOD
         STOP 'COMPLETE'
      ELSE IF (STATUS .EQ. 4) THEN                 ! Parity error
         NWORDS = 2*NDIM
         CALL ACPTBAD(DN, UDA, WC, TERMCND, UBCNT, NWORDS)
         IF (TERMCND .LT. 0) THEN
            PRINT *,'ERROR WHEN ACCEPTING BAD DATA = ',TERMCND
            STOP 'ERROR'
         ENDIF
         NWORDS = WC
      ENDIF
C     ...
C     Process data
C     ...
      GOTO 2000              ! Continue reading
      END
```

Example 2 shows how ACPTBAD works with the Fortran READ statement.

```
      PROGRAM EXAMPLE1
      IMPLICIT INTEGER(A-Z)
      PARAMETER(NBYTES=40000,NDIM=NBYTES/8,DN=99)
      DIMENSION BUFFER(1:NDIM)
      DIMENSION UDA(1:2*NDIM)
      PARAMETER(BADDATA=1356)



2000 CONTINUE
      READ(DN,ERR=3000,IOSTAT=ISTAT,END=5000)BUFFER
C   ...
C   Process data
C   ...
        GOTO 2000                                  ! Continue reading
 3000 CONTINUE
      IF (ISTAT .EQ.  BADDATA) THEN                ! Parity error
         NWORDS = 2*NDIM
         CALL ACPTBAD(DN, UDA, WC, TERMCND, UBCNT, NWORDS)
         IF (TERMCND .LT.  0) THEN
            PRINT *,'ERROR WHEN ACCEPTING BAD DATA = ',TERMCND
            STOP 'ERROR'
         ENDIF
      ELSE
         PRINT *,'ERROR ',ISTAT,' FROM READ'
         STOP 'ERROR'
      ENDIF
C ...
C Process bad data
C ...
      GOTO 2000
 5000 CONTINUE
      STOP 'COMPLETE'
      END
```

SEE ALSO

SKIPBAD(3F)

NAME

CHECKTP – Checks tape position

SYNOPSIS

CALL CHECKTP(*unit*, *istat*, *icbuf*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CHECKTP checks the tape position.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

istat Result of CHECKTP:

- −1 No status
- 0 At EOV
- 1 Tape off reel
- 2 Tape mark detected
- 3 Blank tape detected

icbuf Number of blocks in circular buffer. Always 0 under the UNICOS operating system (included for COS compatibility).

NOTES

The file must be an opened tape or ER90 file and must use the FFIO tape layer.

SEE ALSO

CLOSEV(3F), ENDSP(3F), SETSP(3F), STARTSP(3F)

NAME

ENDSP – Disables special tape processing

SYNOPSIS

CALL ENDSP(*unit*, *istat*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

This routine disables special tape processing.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

istat Result of ENDSP:

- =0 Special processing was successfully disabled.
- ≠0 Error occurred when disabling special processing.

NOTES

The following preconditions must exist:

- The file must be an opened tape file.
- EOV processing must be enabled.
- Special processing must be disabled before you can enable it.
- Special processing must be enabled before you can disable it.

ENDSP removes an alternate path to or from a tape. Tape blocks that were held aside are written to tape.

SEE ALSO

CHECKTP(3F), CLOSEV(3F), SETSP(3F), STARTSP(3F)

NAME

READIBM – Reads two IBM 32-bit floating-point words from each Cray Research 64-bit word

SYNOPSIS

CALL READIBM(*unit*, *fwa*, *word*, *increment*)

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

READIBM reads two IBM 32-bit floating-point words from each Cray Research 64-bit word.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable, expression, or constant. *unit* can be a unit number or file name as a Hollerith constant of 7 characters or less.

fwa First word address (FWA) of the user data area.

word Number of words needed.

increment Increment of the IBM words read.

On exit, the IBM 32-bit format is converted to the equivalent Cray Research 64-bit value. The Cray Research 64-bit words are stored in the user data area.

NOTES

IBM2CRAY(3F) is the recommended routine for performing the same conversion.

SEE ALSO

ACPTBAD(3F), READ(3F), READC(3F), SKIPBAD(3F), WRITE(3F), WRITEC(3F), WRITIBM(3F)

NAME

RNLFLAG, RNLDELM, RNLSEP, RNLREP, RNLCOMM – Manipulates characters recognized by NAMELIST

SYNOPSIS

```
CALL RNLFLAG(char, mode)
CALL RNLDELM(char, mode)
CALL RNLSEP(char, mode)
CALL RNLREP(char, mode)
CALL RNLCOMM(char, mode)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

These routines perform character manipulation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

char For RNLFLAG, an echo character. Default is E.
For RNLDELM, a delimiting character. The defaults are $ and &.
For RNLSEP, a separator character. Default is a comma (,).
For RNLREP, a replacement character. Default is =.
For RNLCOMM, a trailing comment indicator. Defaults are a colon (:) and a semicolon (;).

mode *mode* can have the following values:

=0 Delete character

≠0 Add character

In each of these user-control subroutine argument lists, *char* is a character variable that contains any ASCII character. Except on the CRAY T90 series, *char* may also be a Hollerith character, specified by `1L`*x* or `1R`*x*.

These routines support the Cray Research Fortran 77 (CF77) namelist extension. The ANSI Fortran 90 standard introduced a standardized version of namelist I/O. The facilities provided by these routines are not used by namelist I/O in a Fortran program compiled with Cray Research Fortran 90 (CF90) unless the `-f77` option is supplied on the `assign` command for a file or unit.

`RNLFLAG` adds or removes *char* from the set of characters that, if found in column 1, initiates echoing of the input lines to `stdout`.

`RNLDELM` adds or removes *char* from the set of characters that precede the `NAMELIST` group name and signal end-of-input.

`RNLSEP` adds or removes *char* from the set of characters that must follow each constant to act as a separator.

`RNLREP` adds or removes *char* from the set of characters that occur between the variable name and the value.

`RNLCOMM` adds or removes *char* from the set of characters that initiate trailing comments on a line.

No checks are made to determine the reasonableness, usefulness, or consistency of these changes.

SEE ALSO

`RNLECHO`(3F), `RNLSKIP`(3F), `RNLTYPE`(3F), `WNL`(3), `WNLLINE`(3F), `WNLLONG`(3F)

NAME

RNLECHO – Specifies output unit for NAMELIST error messages and echo lines

SYNOPSIS

```
CALL RNLECHO(unit)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

RNLECHO specifies the output unit for NAMELIST error messages and echo lines.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

unit Output unit to which error messages and echo lines are sent. If *unit*= –1, error messages and lines echoed because of an E in column 1 go to standard output (stdout).

If *unit* ≤ 0, error messages and input lines are echoed to *unit*, regardless of any echo flags present. If *unit*=6 or *unit*=101, stdout is implied.

SEE ALSO

RNL(3), RNLSKIP(3F), RNLTYPE(3F) WNL(3), WNLLINE(3F), WNLLONG(3F)

NAME

RNLSKIP – Takes appropriate action when an undesired NAMELIST group is encountered

SYNOPSIS

CALL RNLSKIP(*mode*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

RNLSKIP determines an action if the NAMELIST group encountered is not the desired group.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

mode *mode* can have one of the following values:

- >0 Skips the record and issues a message (default)
- =0 Skips the record
- <0 Aborts the job or goes to the optional ERR= branch

SEE ALSO

RNL(3), RNLECHO(3F), RNLTYPE(3F), WNL(3), WNLLINE(3F), WNLLONG(3F)

NAME

RNLTYPE – Determines action if type mismatch occurs across equal sign on NAMELIST input record

SYNOPSIS

CALL RNLTYPE(*mode*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

RNLTYPE determines an action if a type mismatch occurs.

mode If *mode*≠0, converts the constant to the variable type (default) of the variable (default). If *mode*=0, aborts the job or goes to the optional ERR= branch.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

SEE ALSO

RNL(3), RNLECHO(3F), RNLSKIP(3F), WNL(3F), WNLLINE(3F), WNLLONG(3F)

NAME

SETSP – Enables and disables EOV processing

SYNOPSIS

CALL SETSP(*unit*, *iflag*, *istat*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SETSP enables and disables end-of-volume (EOV) processing.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

iflag Enable or disable flag; *iflag* has the following arguments:

- =0 Disables EOV processing
- ≠0 Enables EOV processing

istat Result of SETSP; *istat* can have one of the following values:

- =0 No error
- ≠0 Error occurred

NOTES

The following preconditions must exist:

- The file must be an opened tape or ER90 file and must use the FFIO tape layer.
- EOV processing must be disabled for you to enable it.
- EOV processing must be enabled for you to disable it.

SEE ALSO

CHECKTP(3F), CLOSEV(3F), ENDSP(3F), STARTSP(3F)

UNICOS Tape Subsystem User's Guide, publication SG–2051

NAME

SETTP – Positions a tape file at a tape block and/or a tape volume

SYNOPSIS

CALL SETTP(*unit*, *nbs*, *nb*, *nvs*, *nv*, *ivi*, *synch*, *istat*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

SETTP lets a program position a tape file at a particular tape block and/or tape volume. Tape volumes and blocks are numbered sequentially, starting at 1. Volume positioning, if selected, occurs prior to any block positioning. Volumes can be selected either by number or by volume identifier.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

nbs Block number request sign. This parameter must be set to one of the following Hollerith values:

- 1H+ Indicates that the *nb* parameter is the number of blocks to skip forward, relative to the current position.
- 1H– Indicates that the *nb* parameter is the number of blocks to skip backward relative to the current position.

`1H` Indicates that the *nb* parameter is an absolute block number, relative to the last tape mark or beginning-of-volume. If a nonzero value is specified for the *nv* parameter, positioning is absolute with respect to that tape volume. If *nv* is 0, and the `-T` parameter was not present on the `tpmnt`(1) command, the positioning is absolute with respect to the current tape volume. If *nv* is 0, and the `-T` parameter was present on the `tpmnt` command, then positioning is absolute with respect to the last tape mark read or written. One blank space must follow `1H`.

nb Integer block number. Specifies the relative or absolute block number to which the tape file is to be positioned. A block number of 0 indicates no block positioning should be performed. A negative block number is not valid.

nvs Volume number request sign. This parameter must be set to one of the following Hollerith values:

`1H+` Indicates that the *nv* parameter is the number of volumes to skip forward, relative to the current position.

`1H-` Indicates that the *nv* parameter is the number of volumes to skip backward relative to the current position.

`1H` Indicates that the *nv* parameter is an absolute volume number, relative to the beginning of the volume identifier list (specified on the `tpmnt`(1) command). One blank space must follow `1H`.

nv Integer volume number. Specifies the relative or absolute volume to which the tape file is to be positioned. A volume number of 0 indicates no volume positioning is to be performed. A negative volume number is invalid.

ivi Name of volume identifier to be mounted. The name must be left-justified and zero-filled. A nonzero *ivi* parameter is not valid if *nbs* is `1H+` or `1H-`, or if *nvs* is `1H+` or `1H-`, or if *nv* is nonzero. Specifying the volume identifier currently mounted causes it to be rewound.

synch Parameter to allow compatibility with the COS operating system; not supported under the UNICOS operating system. The tape file is always synchronized (that is, all pending I/O is allowed to complete) before any positioning.

istat Integer return status. On exit, this parameter contains a value indicating the status of the positioning request.

- =0 Indicates that the file was successfully positioned.
- ≠0 Indicates that an error was encountered trying to position the tape file, or an argument that was not valid was passed to SETTP (for example, *nb*<0).

NOTES

The file must be a tape or ER90 file and must use the FFIO tape layer.

On Cray MPP systems, SETTP is supported for IBM-compatible tapes with the er90 FFIO layer. This support may be removed in a future release. The er90 layer does not guarantee that records correspond to physical tape blocks. See the assign(1) man page and the *UNICOS Tape Subsystem User's Guide*, publication SG–2051 for more information about the er90 FFIO layer.

SEE ALSO

GETTP(3F)

tpmnt(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

SKIPBAD – Skips bad data

SYNOPSIS

CALL SKIPBAD(*unit, blocks, termcnd*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SKIPBAD lets you skip bad data so that bad data is not sent to the user-specified buffer.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number; *unit* may contain Hollerith data of not more than 7 characters.

blocks On exit, this is the number of physical tape blocks skipped.

termcnd On exit, *termcnd* can have one of the following values:

- <0 Not positioned at end-of-block
- =0 Positioned at end-of-block
- =1 Positioned at end-of-file

EXAMPLES

```
      PROGRAM EXAMPLE1
      IMPLICIT INTEGER(A-Z)
      PARAMETER(NBYTES=40000,NDIM=NBYTES/8,DN=99)
      DIMENSION BUFFER(1:NDIM)
      DIMENSION UDA(1:2*NDIM)

 2000 CONTINUE
      NWORDS = NDIM
      CALL READ(DN,BUFFER,NWORDS,STATUS)
      IF((STATUS.EQ.2) .OR. (STATUS.EQ.3)) THEN    ! EOF or  EOD
         STOP 'COMPLETE'
      ELSE IF (STATUS .EQ. 4) THEN                 ! Parity error
         CALL SKIPBAD(DN, BLOCKS, TERMCND)
         IF (TERMCND .LT. 0) THEN
            PRINT *,'ERROR WHEN SKIPPING BAD DATA = ',TERMCND
            STOP 'ERROR'

                ELSEIF (TERMCND.EQ.0)THEN
                   PRINT *,'SKIPPED ',BLOCKS,' BAD BLOCKS'
                ELSE                                    ! EOF
                   PRINT *,'SKIPPED ',BLOCKS,' BAD BLOCKS'
                   STOP 'COMPLETE'
                ENDIF
                GOTO 2000
             ENDIF
        C    ...
        C    Process data
        C    ...
             GOTO 2000              ! Continue reading
             END
```

SEE ALSO

ACPTBAD(3F)

NAME

SKIPF – Skips files

SYNOPSIS

CALL SKIPF(*iunit*, *ifile*, [*istat*, [*retstat*]])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SKIPF directs the system to skip a specified number of files from the current position.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

iunit An integer variable or array element containing the Fortran unit number. Name of the file or unit number to be positioned.

ifile Integer number. This parameter specifies the number of files to be skipped. If *ifile* is negative, SKIPF positions backwards. If *iunit* is positioned midfile, the partial file skipped counts as one file.

istat A 2-element integer array that returns the number of files skipped in the second element. *istat* is an optional parameter.

retstat Integer number. *retstat* is an optional argument. If present, *retstat* contains a value indicating the status of the SKIPF request on return from the SKIPF subroutine. If this argument is not present and a fatal error occurs while positioning, the program is aborted. *retstat* can have one of the following values:

=0 Indicates that positioning was successful

≠0 Indicates that an error occurred

NOTES

SKIPF does not skip past end-of-data (EOD) or beginning-of-data (BOD). If BOD is encountered before *ifile* files have been skipped when skipping backward, the file is positioned after the BOD. When skipping forward, the file is positioned before the EOD of the current file. SKIPF is currently supported only for online tape or ER90 files that use the FFIO tape layer.

SKIPF can be used for positioning by user tape mark for tapes (see the -T option on the tpmnt(1) command). SKIPF may not be used to position to different files within a multifile volume (see the -q option of the tpmnt(1) command).

SEE ALSO

tpmnt(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

STARTSP – Enables special tape processing

SYNOPSIS

CALL STARTSP(*unit*, *istat*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

STARTSP enables special end of volume (EOV) tape processing.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

istat Result of STARTSP; *istat* can have one of the following values:

=0 Special processing initiated

≠0 Special processing not initiated

NOTES

The following preconditions must exist:

- The file must be an opened tape or ER90 file and must use the FFIO tape layer.
- EOV processing must be enabled.
- Special processing must be disabled.

STARTSP creates an alternative path to or from a tape. Tape blocks in the pipeline are held aside. Subsequent write operations will go directly to tape; subsequent read operations will come directly from tape (if data is available) or from the blocks in the pipeline. Both read and write operations are performed in first-in-first-out (FIFO) order. After you have read from the blocks in the pipeline, they are unavailable for writing after the ENDSP(3F) routine.

SEE ALSO

CHECKTP(3F), CLOSEV(3F), ENDSP(3F), SETSP(3F)

UNICOS Tape Subsystem User's Guide, publication SG–2051

NAME

TSYNC – Requests tape synchronization

SYNOPSIS

CALL TYSNC(*unit*, *istat*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

TSYNC synchronizes a tape or ER90 file. This request is ignored if the last operation was a read. If the last operation was a write, the tape file is synchronized (that is, buffered data is written to the tape). This routine will return an error if the user has specified any FFIO layers other than the tape layer, the er90 layer, or the bufa layer (see assign(1)).

It is not valid to call TSYNC in the following circumstance: when end-of-volume (EOV) processing is enabled, and when the user has reached the EOV but has not started special processing.

If TSYNC is called when EOV processing is enabled, it may cause the user to reach EOV. In this case, TSYNC will return without error, but the tape may not be synchronized. When EOV processing is enabled, the user should check to see if EOV was reached by calling CHECKTP(3F) after calling TSYNC.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

unit An integer variable or array element containing the Fortran unit number or Hollerith data of not more than 7 characters.

istat If *istat*=0, no error occurred. If *istat* is nonzero, an error or warning occurred.

NOTES

The file must be a tape or ER90 file.

SEE ALSO

`CHECKTP`(3F)

`assign`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

WNLFLAG, WNLDELM, WNLSEP, WNLREP – Provides user control of NAMELIST output format

SYNOPSIS

```
CALL WNLFLAG(char)
CALL WNLDELM(char)
CALL WNLSEP(char)
CALL WNLREP(char)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

These routines provide user control of output formatting.

char For WNLFLAG, the first ASCII character of the first line. Default is a blank.
For WNLDELM, a NAMELIST delimiter. Default is &.
For WNLSEP, a NAMELIST separator. Default is a comma (,).
For WNLREP, a NAMELIST replacement character. Default is =.

WNLFLAG changes the character written in column 1 of the first line from blank to *char*. Typically, *char* is used for carriage control if the output is to be listed, or for forcing echoing if the output is to be used as input for NAMELIST reads.

WNLDELM changes the character preceding the group name and END from & to *char*.

WNLSEP changes the separator character immediately following each value from , to *char*.

WNLREP changes the replacement operator that comes between *name* and *value* from = to *char*.

In each of these subroutines, *char* is a character variable that contains any ASCII character. Except on the CRAY T90 series, *char* may also be a Hollerith character, specified by 1L*x* or 1R*x*.

These routines support the Cray Research FORTRAN 77 (CF77) namelist extension. The ANSI Fortran 90 standard introduced a standardized version of namelist I/O. The facilities provided by these routines are not used by namelist I/O in a Fortran program compiled with Cray Research Fortran 90 (CF90) unless the `-f77` option is supplied on the `assign`(1) command for a file or unit.

SEE ALSO

`RNL`(3), `RNLECHO`(3F), `RNLSKIP`(3F), `RNLTYPE`(3F), `WNLLINE`(3F), `WNLLONG`(3F)

NAME

WNLLINE – Allows each NAMELIST variable to begin on a new line

SYNOPSIS

CALL WNLLINE(*value*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WNLLINE allows each NAMELIST variable to begin on a new line.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

value The following values are possible for the *value* argument:

- 0 No new line
- 1 New line for each variable

SEE ALSO

RNL(3), RNLECHO(3F), RNLSKIP(3F), RNLTYPE(3F), WNL(3), WNLLONG(3F)

NAME

WNLLONG – Selects NAMELIST output line length

SYNOPSIS

CALL WNLLONG(*length*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WNLLONG specifies the output line length of the NAMELIST variable.

length The maximum output line length for NAMELIST writes, in the range 8 < *length* < 267. A length of -1 has the effect of setting the output length to the initial, default value (133 for NAMELIST output).

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

NOTES

WNLLONG is overridden by the RECL parameter on the OPEN statement. Where possible, RECL should be used in place of WNLLONG to select the NAMELIST output line length.

SEE ALSO

RNL(3), RNLECHO(3F), RNLSKIP(3F), RNLTYPE(3F), WNL(3), WNLLINE(3F)

NAME

WRITIBM – Writes two IBM 32-bit floating-point words from each Cray 64-bit word

SYNOPSIS

CALL WRITIBM(*unit, fwa, value, increment*)

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

WRITIBM writes two IBM 32-bit floating-point words from each Cray 64-bit word.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

unit	File name or unit number
fwa	First word address (FWA) of the user data area
value	Number of values to be written
increment	Increment of the source (Cray) words written

NOTES

On exit, IBM 32-bit words are written to the unit.

SEE ALSO

ACPTBAD(3F), READ(3F), READC(3F), READIBM(3F), SKIPBAD(3F), WRITE(3F), WRITEC(3F)

NAME

INTRO_FFIO – Describes performance options available with the FFIO layers

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Flexible File I/O (FFIO) system lets the user specify a comma-separated list of layers through which I/O data is to be passed. This is done by providing a value for the *spec* argument to the -F option on the assign(1) command. This specifies a *class* of processing to be done on the data.

All FFIO routines run on all Cray Research systems.

The following C routines are used with FFIO and are documented in this section:

- fffcntl(3C): performs functions on files opened using FFIO
- fflistio(3C): initiates a list of I/O requests using FFIO
- ffopen(3C): opens or closes a file using FFIO
- ffpos(3C): positions files opened using FFIO
- ffread(3C): reads from a file using FFIO
- ffreada(3C): provides asynchronous read using FFIO
- ffseek(3C): repositions a FFIO file
- ffwritea(3C): provides asynchronous write using FFIO
- ffsetsp(3C): initiates EOV processing for files opened using FFIO
- ffwrite(3C): writes to a file using FFIO

Specifying FFIO Layers

The *spec* argument of the -F option of the assign(1) command comprises a list of layers or filters that are used to manipulate the data file as it is being read or written. The available layers include performance options (such as memory-resident and SDS-resident files) and the capability to read and write files in a variety of different vendors' blocking formats.

For more information about FFIO, see the *Application Programmer's I/O Guide*, publication SG–2168.

The *spec* argument can have the following values:

Class **Explanation**

`blankx` or `blx`
Blank compression filter. This layer is used as a filter to compress or decompress blanks in character data. *type* can be one of the following:

`cos` COS-style blank compression (`blxchr`=27 or Ox10)

`ctss` CTSS-style blank compression (`blxchr`=48 or 0x30)

`c205` CYBER 205-style blank compression (`blxchr`=48 or 0x30)

num1 is the decimal value of the ASCII character used as the escape code to control the blank compression (usually `ESC` or 27). *num2* is the decimal value of the ASCII character that is the object of the compression (usually `BLANK` or 32).

`bmx`
Tape I/O. Each logical record requested is a physical tape block. On Cray PVP systems, the *num1* field represents the size of each buffer, in 512-word sectors. The *num2* field represents the number of buffers. Not supported on Cray MPP systems. Files on ER90 volumes that have been mounted in blocked mode may use this layer (see the *UNICOS Tape Subsystem User's Guide*, publication SG–2051, for information about restrictions on record sizes when using ER90 block mode).

`bufa`
Asynchronous buffering layer. The `bufa` layer provides asynchronous buffering. It allows efficient sequential-access I/O. The *num1* field represents the size in 512-word blocks of each buffer. The *num2* field selects the number of buffers to be used.

`c205`
CDC CYBER 205/ETA records. The CYBER 205-type W record can be selected by specifying `c205.w`. The *num1* field is not permitted. The *num2* field represents the size of the working buffer, in bytes, used in record blocking. If any logical records written exceed this size, a major performance penalty can result.

The *type* field should be specified as `w`. The best performance is currently obtained when *num2* is greater than the largest record to be processed plus 16 bytes.

`cache`

Cached file. The cache layer allows efficient random-access I/O, even when file access is clustered in several regions of a file. During reads and writes to the layer, cache buffers frequently must be preempted. The buffer chosen for preemption is always the least recently accessed buffer at the time of preemption.

The options available are `.mem`, which specifies that buffers reside in memory or `.sds`, which specifies that buffers reside in the SDS (`.sds` is not supported on Cray MPP systems). Memory-resident buffers are the default.

The numeric fields are as follows:

num1 is the size in 512-word blocks of each cache page.
num2 selects the number of cache pages to be used.

`cachea`

Asynchronously cached file. This type of processing usually performs well whenever the `cache` layer might be used. In addition, any sequential forward and sequential backward access through the file is detected. When sequential access patterns are detected while reading, asynchronous read-ahead is performed. When writing, selective asynchronous write-behind is performed.

The options available are `.mem`, which specifies that buffers reside in memory, or `.sds`, which specifies that buffers reside in the SDS (`.sds` is not supported on Cray MPP systems). Memory-resident buffers are the default.

The numeric fields are as follows:

- *num1* is the size in 512-word blocks of each cache page.
- *num2* selects the number of cache pages to be used.
- *num3* selects the number of pages to read ahead asynchronously. The default is 0.
- *num4* selects a shared cache number in the range of 1 to 15. If *num4* is 0, a private cache is indicated.

`cdc`
CDC 60-bit format *type*. The supported formats are as follows:

type	**Format**
`iw`	I-type blocks, W-type records
`cw`	C-type blocks, W-type records
`cs`	C-type blocks, S-type records
`cz`	C-type blocks, Z-type records

The *subtype* field accepts one of the following values that indicate the presence of block trailers and other low-level characteristics:

subtype	**Value**
`disk`	Disk type structure, for use with station transfers of CYBER datasets
`i`	NOS internal tape format
`si`	System internal or SCOPE internal tape format
	No numeric fields are accepted.

`cos` or `blocked`
COS blocking. If specified, *type* must be one of the following:

type	**Action**
`sync`	Uses a single buffer in the blocking and deblocking process. I/O is done strictly synchronously.
`async`	Divides the buffer into two parts and uses asynchronous I/O to transfer the blocked data between the buffer(s) and the logical device. When reading, asynchronous read-ahead is performed, and when writing, asynchronous write-behind is performed. To effectively use `async`, the buffer size should be at least twice the record length.
`auto`	Default (if *type* is not specified). Chooses either synchronous or asynchronous behavior depending on the buffer size. If the buffer size is less than 64 blocks, synchronous behavior is selected. If it is greater than or equal to 64 blocks, asynchronous behavior is selected.

For *num1*, enter the desired buffer size in 512-word blocks (for example, `-F cos:42` requests COS blocking and a 42-block buffer). The *num1* value also determines the record size for underlying layers which perform record blocking. The underlying record size is *num1* blocks if in synchronous mode and *num1*/2 or *num1*/2+1 blocks if in asynchronous mode. For an underlying tape layer, the record size is the tape block size.

`er90` ER90 I/O. Each I/O operation results in a corresponding system call. Adjacent records are not delimited from one another. No subfields are accepted. The byte-stream mode of the device must be used. See `tpmnt`(1). On CRAY T3D systems, a subset of the `er90` layer is available. Until the tape layer is supported on CRAY T3D systems, the `er90` layer can be used on the MPP with IBM-compatible tapes. This may not be allowed in future releases. When used with IBM-compatible tapes, the `er90` layer may read and write data differently than the tape layer. The `er90` layer does not guarantee that records correspond to physical tape blocks.

The `er90` layer is not supported on CRAY T3E.

`event` I/O layer monitoring. Deferred on Cray MPP systems.

The `event` layer monitors I/O occurring between two layers on a per-file basis. This layer generates statistics in an ASCII log file; users can specify what type of report is generated. The `event` layer is enabled by default. User do not have to relink their programs to study I/O performance. To generate information, rerun the program with the `event` layer specified on the `assign` command.

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 file. To append information to an existing file, specify a plus sign (+) before the file name.

Enter one of the following for *statlevel*:

value	**Information reported**
`nostat`	No statistical information is reported.
`summary`	Information on event types that occur at least once are reported.

`brief` A one-line summary for layer activities is reported.

`f77` FORTRAN 77/UNIX Fortran record blocking. This is the common blocking format used by most FORTRAN 77 compilers on UNIX systems. Enter one of the following for *type*:

type	**Format**
`nonvax`	Default. Control words in a format common to machines such as the MC68000.
`vax`	VAX format (byte-swapped) control words.

The specification of `vax` or `nonvax` is not relevant to data conversion.

`fd` Connect to a specific file descriptor. Field *num1* is the decimal value of the file descriptor. Classes named `stdin`, `stdout`, and `stderr` exist as alternate names for `fd:0`, `fd:1`, and `fd:2`.

`ibm` Specify one of the following record formats for *type*:

type	**Format**
`f`	Fixed-length record format. For *num1*, enter the logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes; if specified, *num2* must be equal to *num1*.
`fb`	Fixed-length, blocked record format. For *num1*, enter the logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes; *num2* must be an exact multiple of *num1*.
`u`	Undefined record format. For *num1*, enter the maximum record size, in 8-bit bytes.
`v`	Variable length record format. For *num1*, enter the maximum logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes.
`vb`	Variable length blocked record format. For *num1*, enter the maximum logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes.
`vbs`	Variable length blocked, spanned record format. For *num1*, enter the maximum logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes.

No *subtype* is accepted for the `ibm` class record formats. *num1* does not need to be smaller than *num2*.

`mr` or `sds` Memory-resident and SDS-resident files. The `mr` and `sds` layers let you declare a byte-stream file, but the given file should reside in the specified medium as much as possible; this can improve performance.

Enter one of the following for *type*:

type	**Load Specification**
`save`	Default. Loads the file into the specified medium when the file is opened and writes the data back out when closed.
`scr`	Scratch file. Does not attempt to load at open and discards data on close.

Enter one of the following for *subtype*:

subtype	**Overflow Specification**
`ovfl`	Default. Writes excess data that does not fit in the specified medium to the next lower layer.
`novfl`	If data does not fit in the specified medium, subsequent `write`(1) operations fail.

The `mr` and `sds` layers accept numeric fields. Generally, the numeric field specification represents best-efforts values. They are used for tuning purposes and usually do not cause errors if they are not satisfied precisely as specified. The fields are as follows:

Field	**Value**
num1	Initial size of the memory or SDS allocation. Specified in 512-word blocks. Default is 0.
num2	Maximum size of the memory or SDS allocation. Specified in 512-word blocks. Default is $2^{46}-1$.
num3	Increment size of the memory or SDS allocation. Specified in 512-word blocks. This value is used when allocating additional memory or SDS space. The default is 256 for SDS files and 32 for memory resident files.

For example, if the SDS limit for a process is 1000 blocks, and `sds.scr:1500` is specified, an initial SDS allocation of 1000 blocks is used.

Similarly, if `sds:500:10000:1000` was specified, the first request for SDS space would result in the allocation of 500 blocks. If possible, this is allocated contiguously, but it is not guaranteed. If a file size exceeds 500 blocks, an attempt is made to increase the file by an increment of 1000 blocks. If this attempt fails, whatever can be allocated is given to the file. Assuming no other SDS use, this is 500 blocks. At this point, the file assumes an overflow status.

When using memory-resident files, you must ensure that not too much data is written to the file without limiting its size. Unrestrained and unmanaged growth of such a file can cause heap fragmentation and program abort. If this growth has used all available memory, the error message processor may be unable to issue a message. Therefore, such an abort may be difficult to diagnose and correct. It is recommended that the maximum memory size field be specified in all cases. (For example, `mr.scr::10000` would limit such a file to approximately 5 Mwords of main memory.)

NOTE: On CRAY T3D systems, the library does not handle allocation of SDS space from more than one PE. Use of the `-F sds` attribute for files opened from different PEs is not recommended.

The `sds` layer is not supported on CRAY T3E systems.

`null` Syntactic no-op. No optional fields are accepted. `null` may be specified where syntax demands a value, but no function is desired. This does not perform the same function as `/dev/null`.

`nosve` CDC NOS/VE record formats. The following values are allowed for the *type* and *num* fields:

type	**Value**
`d`	ANSI D format (variable-length) records. For *num1*, enter the maximum logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes.
`f`	ANSI F fixed-length records. For *num1*, enter the logical record size in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes; *num2* must be an exact multiple of *num1*.

`s` ANSI S format (segmented) records. For *num1*, enter the maximum logical record size. For *num2*, enter the maximum physical block size used in the lower-level layers.

`u` Undefined records. For *num1*, enter the maximum logical record size, in 8-bit bytes. For *num2*, enter the maximum physical block size in 8-bit bytes.

`v` NOS/VE V format records. The *num1* field is not permitted. For *num2*, specify the size of the working buffer used in record blocking; if any logical records that are written exceed this size, a major performance penalty can result.

`site` Site layer. This option lets a site add custom I/O handlers for specific needs at load time. See the *Application Programmer's I/O Guide*, publication SG–2168, for details.

`stdin`, `stdout`, or `stderr`
Connects to specific file descriptors 0, 1, and 2, respectively. (See the `fd` class.)

`system` Generic system I/O layer. Selects `bmx`, `er90`, or `syscall` as appropriate.

`syscall` System I/O call. Each I/O operation results in a corresponding system call.

`tape` Tape I/O. Each logical record requested is a physical tape block. This is the same as `bmx`. On Cray PVP systems, the *num1* field represents the size of each buffer, in 4096-byte blocks. The *num2* field represents the number of buffers. Not supported on Cray MPP systems.

`text` Special character terminated records. Enter one of the following for *type*:

type	**Format**
`nl`	Newline-separated records
`eof`	Newline-separated records and the special character sequence `~e` on a line by itself delimiting EOF
`205`	CYBER 205-style text file
`ctss`	CTSS-style text format

This class accepts *num* specifications. If specified, *num1* represents the decimal value of the ASCII character to use to delimit records; the default varies depending on the *type*. For *num2*, enter the requested working buffer size in bytes.

`user` User layer. This option allows a user to add custom I/O handlers for specific needs at load time. See the *Application Programmer's I/O Guide*, publication SG–2168, for details.

`vms` For *type*, enter one of the following record formats:

type	**Format**
`f`	Fixed-length records
`v`	Variable length records
`s`	Segmented variable records

Each *type* accepts the following *subtype*s to specify the blocking format within the record type:

subtype	**Format**
`tape`	ANSI standard record format. This *subtype* should be used with labeled VAX/VMS tapes.
`bb`	Binary blocked format. This *subtype* should be used with files that are to be fetched or disposed with a `BB` or `TB` format, or with unlabeled magnetic tapes. This *subtype* requires an enclosing blocking; for example, `vms.s.bb,bmx` or `vms.s.bb,cos`.
`tr`	Transparent format. This *subtype* should be used with files that are transferred between the VAX/VMS system by using `fetch` or `dispose` and the `TR` format. Any other method that precisely transfers the disk image, including all VMS control words, will also work. Note that `ftp` does not correctly transfer nontext, variable-length record files.

This class accepts *num1* and *num2* fields; they have a similar meaning to `ibm` class. For *type* `s`, *num1* is ignored. For *type* `f`, *num2* need not be a multiple of *num1*.

EXAMPLES

The following are example FFIO specifications.

Example 1: The following example specifies FORTRAN 77/UNICOS Fortran blocking with a working buffer of 128,000 bytes to allow efficient creation of logical records up to 127,992 bytes:

```
f77::128000
```

Example 2: The following example specifies IBM VBS records with a block size of 16,384 bytes:

```
ibm.vbs::16384
```

Example 3: The following example specifies a memory-resident layer with an initial memory size of 100 512-word blocks:

```
mr:100
```

Example 4: The following example specifies VMS V records with a maximum record size of 1000 bytes:

```
vms.v.tr:1000
```

Example 5: The following example specifies the SS-resident layer that loads from disk on open and saves to disk on close, does not allow overflow, has an initial size of 10 512-word blocks in SDS, grows to a maximum size of 559 blocks in minimum increments of 37 blocks:

```
sds.save.novfl:10:559:37
```

See the *Application Programmer's I/O Guide*, publication SG–2168, for more detailed information about FFIO specifications.

NOTES

Some FFIO specification requirements are not obvious. For example, CYBER 205 R-type records must be requested with `blx.ctss,text.205`.

The Fortran I/O library checks for conflicting attributes when file name and unit attributes are both present during `OPEN` processing for a Fortran unit. The existence of an assign attribute for both the file and the unit results in an error condition.

SEE ALSO

acptbad(3F), assign(3F), ffopen(3C), openms(3F), opendr (see openms(3F)), skipbad(3F)

df(1), ln(1), setf(1), tpmnt(1), write(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

asgcmd(1), assign(1) in the *Application Programmer's I/O Guide*, publication SG–2168

ialloc(2), open(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

SDSALLOC(3F) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

`fffcntl` – Performs functions on files opened using flexible file I/O

SYNOPSIS

```
#include <ffio.h>
int fffcntl (int fd, int cmd, long *arg, struct ffsw *stat);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The `fffcntl` function performs a variety of functions on files opened by `ffopen` or `ffopens`, using flexible file I/O (FFIO). Arguments are as follows:

fd Number returned by function `ffopen` or `ffopens`.

cmd Specifies values defined in header file `foreign.h`. See under the next heading, VALUES FOR *cmd* ARGUMENT.

arg The type and value of *arg* is *cmd* specific. See preceding descriptions.

stat Pointer to the status return structure.

Values for *cmd* Argument

The following values can be used for the *cmd* argument introduced above:

`FC_ACPTBAD`

Accepts bad data (valid only for online tape files and `er90` files on Cray PVP systems). *arg* is a pointer to structure `ffc_baddata_s`, defined in header file `foreign.h`. This *cmd* is valid with the following FFIO layers: `tape`. The fields of this structure have the following meaning:

`ffc_bytes` Number of bytes of bad data transferred is returned in this field.

`ffc_maxflag` Set this value to 1 if a maximum value is specified in field `ffc_maxwords`. Set this value to 0 if no maximum value is specified.

`ffc_maxwords` Maximum number of words of bad data to transfer to user's data area. If the number of words of bad data in the block exceed this value, the excess is discarded.

`ffc_termcnd` Position after accepting bad data is shown in this field, as follows: 0 indicates end of block; 1 indicates `EOF`; 2 indicates `EOD`; and a value less than 0 indicates that an error occurred. The absolute value is the error number.

`ffc_uda` User data area to receive bad data.

`FC_ASPOLL`

Checks for completion of an asynchronous FFIO request. Argument *arg* is a pointer to a structure of type `struct ffsw`, which had previously been passed to an asynchronous FFIO request. The purpose of this call is to pass control to the underlying layers to do intermediate processing or cleanup on the request. If the request is complete, fields in the status return structure are set as described under function `ffreada`(3C).

`FC_CHECKTP`

Checks tape position. This value is valid only for online tape files and `er90` files on Cray PVP systems. *arg* is a pointer to structure `ffc_chktp_s`, defined in header file `foreign.h`. This command is valid with the following FFIO layers: `tape`. The fields of this structure have the following meaning:

stat The status of the tape, as follows:

−1 = No status
0 = At EOV
1 = Tape off reel
2 = Tape mark detected
3 = Blank tape detected

The remaining fields are unused.

`FC_ENDSP`

Ends special processing. This value is valid only for online tape files and `er90` files on Cray PVP systems. *arg* is unused. This function removes the alternate path to tape created by `FC_STARTSP`. Tape blocks that were held aside are written to tape. This *cmd* is valid with the following FFIO layers: `tape`.

FC_CLOSEV
: Closes volume and mounts next volume in the Volume Identifier list. *arg* is unused. This value is valid only for online tape files and er90 files on Cray PVP systems. This *cmd* is valid with the following FFIO layers: tape, er90, bufa.

FC_GETTP
: Retrieves information about an opened tape file (valid only for online tape files and er90 files on Cray PVP systems). *arg* is a pointer to structure ffc_gettp_s, defined in header file foreign.h.

  On Cray MPP systems, FP_GETTP is supported for IBM-compatible tapes with the er90 FFIO layer. This support may be removed in a future release. The er90 layer does not guarantee that records correspond to physical tape blocks. See the assign(1) man page and the *UNICOS Tape Subsystem User's Guide*, publication SG–2051 for more information about the er90 FFIO layer.

  The fields of this structure have the following meaning:

  ffc_glen — Number of words to copy to the array pointed to by field ffc_pa.

  ffc_synch — Synchronization value, as follows: a value of 1 indicates to synchronize the dataset before obtaining position information; a value of 0 means do not synchronize the dataset. This field is ignored if the last operation was a read. It is also invalid to specify this value if end-of-volume (EOV) processing is enabled, and the user has reached EOV but has not started special processing.

  ffc_pa — Address of array that will contain information returned by this function. The values returned in this array are as follows:

  ffc_pa[0] — Current volume identifier.

  ffc_pa[1] – ffc_pa[6] — Characters 1–48 of the path name of the file opened to this tape.

  ffc_pa[7] — Integer file section number.

  ffc_pa[8] — Integer file sequence number.

  ffc_pa[9] — Integer block number relative to tape mark specified in ffc_pa[22].

`ffc_pa[10]` Integer number of blocks in the library buffer. If additional processing layers have been specified with `assign`(1) or `asgcmd`(1), those layers may also hold buffered data, but they will not be included in this field.

`ffc_pa[11]` Integer number of blocks in the IOP or system buffer.

`ffc_pa[12]` Integer device ID or unit number.

`ffc_pa[13]` Device identifier or name.

`ffc_pa[14]` Generic device name.

`ffc_pa[15]` Last device function.

`ffc_pa[16]` Last device status.

`ffc_pa[17]` Data transfer count in bytes.

`ffc_pa[18]` Buffer memory sector count.

`ffc_pa[19]` Partial block bytes in buffer memory.

`ffc_pa[20]` Outstanding sector count.

`ffc_pa[21]` Outstanding block count.

`ffc_pa[22]` User tape mark number, including tape marks embedded in the data.

`ffc_pa[23]` Direction from tape mark in previous word: 0 = after tape mark; 1 = before tape mark.

`ffc_pa[24]` Today's year modulus 100.

`ffc_pa[25]` Today's Julian day.

`ffc_pa[26]` File identifier, up to the first 8 characters.

`ffc_pa[27]` Record format name.

`ffc_pa[28]` Tape density: 1 = 1600 bpi; 2 = 6250 bpi.

`ffc_pa[29]` Maximum block size.

`ffc_pa[30]` Record length.

`ffc_pa[31]` File status: 1 = new; 2 = old; 3 = append.

`ffc_pa[32]` Label type: 1 = no label; 2 = ANSI label; 3 = IBM standard label; 4 = bypass label.

`ffc_pa[33]` Integer file sequence number of first file on volume.

`ffc_pa[34]` Ring status: 0 = ring out; 1 = ring in.

`ffc_pa[35]` Expiration year modulus 100.

`ffc_pa[36]` Expiration Julian day.

`ffc_pa[37]` First volume identifier of file.

`ffc_pa[38]` User end-of-volume status: 0 = EOV processing off; 1 = EOV processing on.

`ffc_pa[39]` User end-of-volume processing status: 0 = not in active EOV processing; 1 = in active EOV processing.

`ffc_pa[40]` User read/write tape mark status: 0 = user read/write tape mark not allowed;
1 = user read/write tape mark is allowed.

`ffc_pa[41]` Block attribute: `'B'` = blocked records; `'S'` = spanned records, if the record format is `'V'`, or standard records, if the record format is `'F'`; `'R'` = blocked and spanned records, if the record format is `'V'`, blocked and standard records, if the record format is `'F'`; `' '` = none of the previous values.

`ffc_pa[42] – ffc_pa[47]`
File identifier.

`FC_GETINFO`
Gets information about the layers connected to this open file. *arg* is a pointer to structure `ffc_info_s`. The information returned in this structure is as follows:

`ffc_flags`
Flag word containing attributes of the file/connection. These bit masks are defined in header file `foreign.h` and are set if true, as follows:

`FFC_STRM`	Can handle stream I/O.
`FFC_REC`	Can handle records.
`FFC_WEOF`	Can represent `EOF`.
`FFC_WEOD`	Can represent `EOD` (always set).
`FFC_BKSP`	Can handle backspace.
`FFC_BKFIL`	Can handle backfile.
`FFC_SEEKA`	Can seek absolute.
`FFC_SEEKR`	Can seek relative.

FFC_SEEKE	Can seek to end.
FFC_POSREC	Can position by record number.
FFC_POSFIL	Can position by EOF mark.
FFC_RWND	Can rewind by seek(x,0,0).
FFC_FIXD	Can do fixed-length records.
FFC_VAR	Can do variable-length records.
FFC_BINARY	Can do binary data.
FFC_CODED	Can do formatted (character) data.
FFC_RDM	Can do random I/O (no truncation).
FFC_SEQ	Can do sequential I/O.
FFC_ASYNC	Can do asynchronous I/O. (All layers have asynchronous entry points, but this bit tells whether the behavior is actually async.)
FFC_WRTRUNC	Write implies truncation.
FFC_NOTRN	Does no transformation on data; no control words are added or subtracted. Data is not changed.

ffc_gran
: Minimum granularity. This is the smallest size in bits of a valid data transfer. For example, the system call layer has an ffc_gran of 8, as it can handle a byte as its smallest unit of data transfer. Some CDC record formats have a granularity of 60.

ffc_reclen
: Valid only for fixed length records. This is the record length in bits.

`ffc_fd`
: Lowest level file descriptor for the layer that makes system calls. This is not always available, or may not be meaningful for some layers or combinations of layers. This is -1 if no descriptor is available.

`FC_RECALL`
: Awaits completion of an asynchronous FFIO request. Argument *arg* is a pointer to a structure of type `struct ffsw`, which is the status return structure of the asynchronous request. Function `fffcntl` waits for completion of the asynchronous request, if necessary. Fields in the status return structure are set as described under function `ffreada`(3C).

`FC_SCRATCH`
: Specifies that a file is to be deleted at close time. The *arg* argument is a pointer to *int*. On exit, **arg* is set to contain zero or more of the following result bits:

`SCR_NOFLUSH`	Set if `ffclose` processing has been optimized to suppress buffer flushing.
`SCR_SINGLELINK`	Set if the file is not a pipe or a tty, has a link count equal to one, and is not a symbolicly linked file.
`SCR_UNLINKED`	Set if this `fffcntl` call has successfully unlinked the file.

`FC_SKIPBAD`
: Skips bad data (valid only for online tape files and `er90` files on Cray PVP systems). *arg* is a pointer to structure `ffc_baddata_s`, defined in header file `foreign.h`. This *cmd* is valid with the following FFIO layers: `tape`. The fields of this structure used by `FC_SKIPBAD` are as follows; all other fields are unused:

`ffc_blocks`	The number of blocks skipped is returned in this field.
`ffc_termcnd`	Position after skipping bad data is returned in this field, as follows: 0 indicates end of block; 1 indicates `EOF` or `EOD`; and a value less than 0 indicates that an error occurred. The absolute value is the error number.

FC_STAT
: Returns a structure much like the one returned by the `fstat` (see `stat`(2)) system call. *arg* is a pointer to a `ffc_stat_s` structure (from `ffio.h`). Fields in this structure are filled in as appropriate by the layers. For the system call layer, all fields are simply retrieved by doing an `fstat` call. For other layers, such as `mr`, the *size* field in the `stat` structure is modified to reflect the buffered data. Other layers can make similar changes to the basic information from the system for similar reasons. The result is a `stat` structure that can be used in the same way and for the same purposes as the `fstat` system call.

FC_SETSP
: Disables special EOV processing. This value is valid only for online tape files and `er90` files on Cray PVP systems. *arg* is an integer value that should be set to 0. See `ffsetsp`(3C) for a description of how to enable special EOV processing. This *cmd* is valid with the following FFIO layers: `tape`.

FC_STARTSP
: Starts special EOV processing. This value is valid only for online tape files and `er90` files on Cray PVP systems. *arg* is unused. EOV processing must be enabled prior to starting special EOV processing. This function creates an alternative path to or from a tape. Tape blocks in the pipeline are held aside. Subsequent write operations will go directly to tape; subsequent read operations will come directly from tape (if data is available) or from the blocks in the pipeline. Both read and write operations are performed in FIFO order. After you have read from the blocks in the pipeline, they are unavailable for writing. This *cmd* is valid with the following FFIO layers: `tape`.

FC_TPC_SDBSZ
: Changes the data block size on an ER90 device. This is valid only when using the tape layer. *arg* is the requested new block size. This *cmd* has no effect when it is used with an IBM-compatible tape.

FC_TSYNC
(Valid only for online tape files and er90 files on Cray PVP systems and Cray MPP systems)). It requests that the tape file be synchronized. This command is ignored if the last operation was a read. It is also invalid to request synchronization if the end-of-volume (EOV) processing is enabled, and the user has reached EOV but has not started special processing. If the end-of-volume processing is enabled, the user should check to see if EOV was reached after requesting FC_TSYNC (see the description for FC_CHECKTP). In this case, the fffcntl returns without error, but the tape may not be synchronized (that is, data may remain buffered). This *cmd* is valid with the following FFIO layers: tape, er90, bufa.

RETURN VALUES

The fffcntl function returns 0 on success. Otherwise, it returns -1 and the sw_error field of the stat structure contains the error number.

SEE ALSO

fflistio(3C) ffopen(3C), ffreada(3C), ffsetsp(3C), ffwritea(3C)

assign(1), asgcmd(1), in the *UNICOS User Commands Reference Manual*, publication SR–2011

UNICOS Tape Subsystem User's Guide, publication SG–2051, for more information about the er90 FFIO layer

NAME

`ffiolock`, `ffiounlock` – Provide locking for FFIO functions

SYNOPSIS

```
#include <ffio.h>
int ffiolock (int fd, struct ffsw *stat);
int ffiounlock (int fd [, struct ffsw *stat]);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The `ffiolock` and `ffiounlock` functions provide a way of locking a group of Flexible File I/O (FFIO) function calls that use the same *fd*. Function `ffiolock` sets a lock; any other task that issues an FFIO call with the same *fd* will wait until the lock is released. Function `ffiounlock` releases the lock.

The following arguments are available for this routine:

fd Integer returned by `ffopen` or `ffopens`

stat Status structure

RETURN VALUES

In case of error, both functions return -1 and the `ffsw.sw_error` field contains the error value. Otherwise, 0 is returned.

NAME

`fflistio` – Initiates a list of I/O requests using flexible file I/O

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iosw.h>
#include <sys/listio.h>
#include <ffio.h>

int fflistio (int cmd, struct fflistreq *list, int nent
[,struct ffstat *stat]);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `fflistio` function provides a means to initiate a list of distinct I/O requests and, optionally, waits for all of them to complete. Each I/O request in the list provides for maximum control over the desired I/O characteristics. As much as possible, the operation of this function is the same as the `listio` system call. The purpose of this function is to provide the functionality of `listio`(2) combined with the facilities provided by flexible file I/O (FFIO).

The arguments are as follows:

cmd Any of the following commands:

- `LC_START` Initiates the I/O requests and returns control as soon as possible
- `LC_WAIT` Initiates the I/O requests and returns when all requests have completed

list Pointer to an array of `fflistreq` structures.

nent The number of requests in the list to process.

The `fflistreq` structure includes the following members:

`int li_opcode`	Operation code for the request: `LO_READ` for read and `LO_WRITE` for write.
`unsigned li_drvr:32`	Driver dependent; not used.

`unsigned li_flags:32`	Special request flags: `LF_LSEEK` to set initial file offset to `li_offset`.
`long li_offset`	Initial file byte offset, if `LF_LSEEK` is set in flags.
`int li_fildes`	`fdinfo` pointer, obtained from `ffopen`(3C) or `ffopens`(3C).
`char *li_buf`	Pointer to an I/O data buffer in memory.
`unsigned li_nbyte`	Number of bytes to read or write for each stride.
`struct ffsw *li_status`	Pointer to an I/O status structure where the FFIO functions will put completion status for this request. See `fffcntl`(3C).
`int li_signo`	This value must be zero. Signals are not supported.
`int li_nstride`	Number of strides; defaults to 1.
`long li_filstride`	File stride in bytes; default is for contiguous data flow to/from the file.
`long li_memstride`	Memory stride in bytes; default is for contiguous data flow to/from the memory buffer.

When reading or writing an *n*-dimensional array on a disk, the desired data I/O occurs at regular intervals, but it may not be contiguous. The last three variables in the `fflistreq` structure can be used to specify a compound request, causing multiple sections of data to be transferred. The distance from the start of one section of data on disk to the start of the next section is called the *file stride*. There is an analogous stride through memory.

The operation of the stride parameters are the same as `listio`(2). See `listio`(2) for more information on the details of operation.

When a particular request is complete, the associated `ffsw` structure is filled in whenever an `fffcntl` function is given an opportunity with an `FC_RECALL` or `FC_ASPOLL` function code. In the `ffsw` structure, *sw_stat* is set to a non-zero value upon completion, *sw_error* may contain a system or library error number, and *sw_count* contains the number of bytes actually moved. For a successful compound request, *sw_count* would be *li_nstride * li_nbyte*.

If one or more of the I/O requests are ill-formed and cannot be started, an `LC_WAIT` type command returns immediately.

RETURN VALUES

If `fflistio` completes successfully, a nonnegative integer is returned, indicating the number of requests that were successfully started. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

ERRORS

The `fflistio` system call fails on a variety of conditions including the set of errors returned by `listio`(2) and those returned by the FFIO functions.

EXAMPLES

The following example shows how to use the `fflistio` system call to initiate a list of input requests. The `fflistio` request reads every tenth block (that is, blocks 1, 11, 21, 31, and so on) from file `datafile`. `fffcntl`(3C) is used to determine when the request is complete.

The request is initiated asynchronously (`LC_START`), and the data is stored in the buffer `buf` contiguously. Since input is performed asynchronously, the program can complete other work in parallel with the request.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/iosw.h>
#include <sys/listio.h>
#include <ffio>

#define BLK_SIZ 4096
struct blk {
     char    blk_data[BLK_SIZ];
};

main()
{
    struct fflistreq request;
    struct ffsw reqstat, dumstat;
    struct blk buf[10];
    int fd;

    if ((fd = ffopen("datafile", O_RDONLY)) == -1) {
         perror("open (datafile) failed");
         exit(1);
    }
```

```
    memset (&reqstat, 0, sizeof(reqstat));
    memset (&request, 0, sizeof (request));

    /* Set up the I/O request */
    request.li_opcode = LO_READ;          /*request read*/
    request.li_fildes = fd;               /*file descriptor*/
    request.li_buf = (char *) buf;        /*store input data here*/
    request.li_nbyte = BLK_SIZ;           /*each stride = 1 block*/
    request.li_status = &reqstat;         /*status for this request*/
    request.li_signo = 0;                 /*do not send signal upon completion*/
    request.li_nstride = 10;              /*read 10 strides*/
    request.li_memstride = 0;             /*default (contiguous) memstride*/
    request.li_filstride = 10 * BLK_SIZ;  /*file stride=10 blocks*/

    if (fflistio(LC_START, &request, 1) != 1) {
         perror("fflistio failed");
         exit(1);
    }

    /* other work can be performed here while fflistio request completes */

    fffcntl(fd, FC_RECALL, &reqstat, &dumstat); /* waits for I/O completion */

    printf("Number of bytes read = %d\n\n", reqstat.sw_count);
}
```

SEE ALSO

`fffcntl`(3C), `ffopen`(3C)

`listio`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

ffopen, ffopens, ffclose – Opens or closes a file using flexible file I/O

SYNOPSIS

```
#include <fcntl.h>
#include <ffio.h>

int ffopen (const char *name, int oflag [, int mode [, long
cbits [, struct ffsw *stat]]]);

int ffopens (const char *name, int oflag, int mode, long cbits,
[int cblks,] struct ffsw *stat, const char *str);

int ffclose (int fd [, struct ffsw *stat]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The ffopen and ffopens functions open a file using flexible file I/O (FFIO). The ffopen and ffopens functions are modeled after the open(2) system call. The file associated with *name* is opened, and appropriate structures are built to do special handling on the file. For ffopen, the processing layers to be used are as requested with the asgcmd(1) or assign(1) commands. For ffopens, the layers are specified by the string at *str*.

The ffopen function returns an integer. Although it is not a file descriptor, it is used in much the same way when passed to the other functions such as ffread, ffwrite, or ffclose.

The *oflag*, *mode*, and *cbits* parameters are exactly as in open(2).

The *stat* parameter is a pointer to a status return structure containing a flag, an error indication, a transfer count, and an indication of the condition that terminated the request.

The `ffclose` function closes the file associated with *fd*. It does any necessary flushing and termination for that file. Files opened using `ffopen`(3C) are not guaranteed to be flushed at program termination; call `ffclose` to ensure this.

RETURN VALUES

Upon successful completion, a non-negative integer is returned; this integer is currently a pointer to a control block. Otherwise, -1 is returned, and, if the *stat* parameter is passed, the error value is found in `stat.sw_error`. If the *stat* parameter is not provided, the error code is found in `errno`. The *stat* parameter is required for `ffopens`().

EXAMPLES

To write a C program that will read the records of a COS blocked file named `indata` and copy the data from the file (without blocking information) to `stdout`, compile the following program:

```
#include <stdio.h>
#include <fcntl.h>
#include <ffio.h>

#define BSZ 10000

main()
{
      int fd, ret;
      char buf[BSZ];
      struct ffsw stat;

      fd = ffopen("indata", O_RDONLY, 0, 0, &stat);
      do
      {
            ret = ffread(fd, buf, BSZ, &stat, PARTIAL);
            fwrite(buf, 1, ret, stdout);
       } while (FFSTAT(stat) != FFEOD);
       ret = ffclose(fd);
}
```

Use the following commands to compile, link, and run this program:

```
$ cc cpy.c
$ eval `asgcmd -F cos indata` # declare "indata" is COS blocked
$ a.out
```

The same example program could be written to use the `ffopens` entry point. The layers to use in decoding the blocking information are specified in the call, and not in the shell command `asgcmd`(1), as follows:

```
#include <stdio.h>
#include <fcntl.h>
#include <ffio.h>

#define BSZ 10000

main()
{
      int fd, ret;
      char buf[BSZ];
      char *str;
      struct ffsw stat;

      /* Set up to process COS blocked file. */
      str = "cos";
      fd = ffopens("indata", O_RDONLY, 0, 0, &stat, str);
      do
      {
            ret = ffread(fd, buf, BSZ, &stat, PARTIAL);
            fwrite(buf, 1, ret, stdout);
      } while (FFSTAT(stat) != FFEOD);
      ret = ffclose(fd);
}
```

Use the following commands to compile, link, and run this program:

```
$ cc cpy.c
$ a.out
```

SEE ALSO

`ffread`(3C), `ffseek`(3C)

`asgcmd`(1) in the *Application Programmer's I/O Guide*, publication SG–2168

`open`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

Application Programmer's I/O Guide, publication SG–2168

NAME

ffpos – Positions files opened using flexible file I/O

SYNOPSIS

```
#include <ffio.h>

int ffpos (int fd, int cmd, long *arg , int len, struct ffsw
*stat);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The ffpos function provides a way of positioning files opened by ffopen or ffopens (see ffopen(3C)), using flexible file I/O (FFIO). Arguments are as follows:

fd Value returned by ffopen or ffopens.

cmd Specifies a value defined in header file ffio.h. See the following heading, VALUES FOR *cmd* ARGUMENT.

arg The type and value of this argument is *cmd* specific. See preceding descriptions.

len If *cmd* = FP_GETPOS or FP_SETPOS, *len* specifies the number of Cray words in *arg*. This parameter is ignored for other values of *cmd*.

stat Pointer to the status return structure.

Values for *cmd* Argument

The following values can be used for the *cmd* argument introduced above.

FP_BSEEK

Sets the current file position as specified by *arg and *(*arg+1). *arg contains the bit position requested; this must be a byte boundary. *(*arg+1) specifies one of the following values, defined in header file stdio.h:

0 or SEEK_SET

Sets the pointer to the value of *arg, which must be a non-negative integer.

1 or SEEK_CUR
: Sets the pointer to the current position, plus or minus **arg*. This is supported only in layers that are not record-oriented; layers are specified to `assign -F` as follows: `syscall`, `sds`, `mr` (memory resident), `cache`, `cachea`, `bufa`, `er90`.

2 or SEEK_END
: Sets the pointer to the end of the file, minus **arg*. **arg* must be a non-negative integer. Not all layers support this option.

FP_GABS
: Returns information about the current position in arg, which can later be used by a call to `ffpos` with *cmd* = FP_SABS. *arg* is a pointer to structure `fp_abs` (defined in header file `ffio.h`).

  This call is useful only for online tape and ER90 files. It may be used with the `tape`, `er90`, or `bufa` layers on Cray PVP systems. For some devices, some of the fields in `fp_abs` may be unused. When this request is made after a write function, the library and tape driver flush all remaining data from the write behind to the tape before attempting to get the position. When this request is made after a read request, an approximate address is returned, since there may be data buffered in the controller, system buffers, or library buffers. FP_GABS does not return information about the volume serial number (VSN) currently in use.

FP_GETPOS
: Returns information about the current position in *arg*, which can later be used by a call to `ffpos` with *cmd* =FP_SETPOS. For online tape files, 2 Cray words are returned in *arg*. For files assigned with `assign -F er90`, 4 Cray words are returned in *arg*. For files assigned with `-F cos,er90`, 6 Cray words are returned in *arg*. For all other file types, 1 Cray word is returned in *arg*. For ER90 files, the information returned does not include the VSN. When using the information returned in a call to `ffpos` with *cmd*=FP_SETPOS, you must ensure that you are positioned on the correct volume. This command is unsupported for ER90 files that use the FFIO tape layer. Available only on Cray PVP systems.

FP_SABS
: Sets the position as specified in *arg*. The information in *arg* should have been obtained by a previous call to ffpos with *cmd* = FP_GABS. *arg* is a pointer to structure fp_abs. This call is useful only for online tape and ER90 files. FP_SABS assumes that you are currently positioned on the correct VSN. You must have permission to set the position to the specified address when using absolute track address positioning. Tape manager permission is required for IBM-compatible tapes. For ER90 files, tape manager permission is required to position outside the current partition. Bypass-label permission or tape manager permission is required to position outside the current file, or past a user tape mark. Available only on Cray PVP systems.

FP_SETPOS
: Sets the position as specified by the information in *arg*. The information in *arg* should have been obtained by a previous call to ffpos with *cmd* = FP_GETPOS.

FP_SKIPF
: This value is valid only for online tape files and ER90 files on Cray PVP systems. It directs the system to skip a specified number of files from the current position. The file will not be positioned beyond BOD or EOD. This is used for positioning by user tapemark (see the -T option on the tpmnt(1) command). It may not be used to position to different files within a multifile volume (see the -q option of the tpmnt(1) command. *arg* is a pointer to structure ffp_skipf_s (also defined in header file ffio.h). This *cmd* is valid with the following FFIO layers: tape. The fields of this structure have the following meaning:

  ffp_nfil — On input, specifies the number of files to skip. If the value is negative, the file is positioned backward. On output, contains the number of files skipped.

  ffp_nrec — Currently unused.

FP_SKIPTPMK
: This value is valid only for online tape files on Cray PVP systems. It directs the system to skip a specified number of tape marks from the current position. The file will not be positioned beyond BOD or EOD. This is used for positioning by user tapemark (see the -T option on the tpmnt(1) command). arg is a pointer to structure ffp_skiptpmk_s (defined in <ffio.h>). The fields in this structure have the following meaning:

`ffp_ntpmk` On input, specifies the number of tape marks to skip. If the value is negative, the file is positioned backwards. On output, this field contains the number of tape marks left to position.

`unused1` This field is reserved.

The `FP_SKIPTPMK` *cmd* functions differently from `FP_SKIPF`. Except in the case where you request positioning past EOD, `FP_SKIPF` will position you at the beginning of a file; that is, `FP_SKIPF` will position you either at BOD, immediately after a user tape mark, or at EOD. `FP_SKIPTPMK` functions like the ioctl `TR_PTMS` described in the *UNICOS Tape Subsystem User's Guide*, publication SG–2051. It skips the specified number of tape marks. If skipping forward, it will position you directly after a user tape mark or at EOD. If skipping backwards, it will position you directly before a user tape mark or at BOD. This *cmd* is valid with the following FFIO layers: `tape`.

`FP_SETTP`

arg is a pointer to structure `ffp_settp_s` (also defined in header file `ffio.h`). This *cmd* is valid with the following FFIO layers: `tape`.

On Cray MPP systems, `FP_SETTP` is supported for IBM-compatible tapes with the `er90` FFIO layer. This support may be removed in a future release. The `er90` layer does not guarantee that records correspond to physical tape blocks. See the `assign`(1) man page and the *UNICOS Tape Subsystem User's Guide*, publication SG–2051 for more information about the `er90` FFIO layer.

On input, the fields of `ffp_settp_s` have the following meaning:

`ffp_nb`
Number of blocks to position; should always be a positive number.

`ffp_nbs_p`
Indicates the direction of block positioning. The following values are defined in header <`ffio.h`>:

`FP_TPOS_BACK`
Indicates that the `ffp_nb` field is the number of blocks to skip backward, relative to the current position.

`FP_TPOS_FORW`
Indicates that the `ffp_nb` field is the number of blocks to skip forward, relative to the current position.

FP_TPOS_ABS
Indicates that the ffp_nb field is an absolute block number, relative to the last tape mark number or beginning of the volume. If a nonzero value is specified for the ffp_nv field, positioning is absolute with respect to that tape volume. If ffp_nv is 0, and the -T option was not present on the tpmnt(1) command, the position is absolute with respect to the current tape volume. If ffp_nv is 0, and the -T option was present on the tpmnt command, then positioning is absolute with respect to the last tape mark read or written.

ffp_nv
Number of volumes to position; should always be a positive number. A volume number of 0 indicates no volume positioning is to be performed.

ffp_nvs_p
Indicates the direction of volume positioning. The following values are defined in header file ffio.h:

FP_TPOS_FORW	The number of volumes to skip forward, relative to the current position.
FP_TPOS_BACK	The number of volumes to skip backward, relative to the current position.
FP_TPOS_ABS	Specifies an absolute volume number, relative to the beginning of the volume identifier list (specified on the tpmnt(1) command.)

ffp_vi
Name of volume identifier to be mounted. A nonzero ffp_vi field is invalid if ffp_nbs_p is FP_TPOS_FORW or FP_TPOS_BACK or if ffp_nvs is FP_TPOS_FORW or FP_TPOS_BACK and ffp_nv is nonzero.

RETURN VALUES

If *cmd* is FP_BSEEK, ffpos returns the new bit position of the file on success. For other values of *cmd*, the ffpos function returns 0 on success. On failure, it returns -1 and the sw_error field of the stat structure contains the error number.

SEE ALSO

`ffopen`(3C)

`tpmnt`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

UNICOS Tape Subsystem User's Guide, publication SG–2051

NAME

ffread, ffwrite, ffweof, ffweod – Provides flexible file I/O

SYNOPSIS

```
#include <ffio.h>

int ffread (int fd, char *buf, int nb [, struct ffsw *stat
[, int fulp [, int *ubc]]]);

int ffwrite (int fd, char *buf, int nb [, struct ffsw *stat
[, int fulp [, int *ubc]]]);

int ffweof (int fd [, struct ffsw *stat]);

int ffweod (int fd [, struct ffsw *stat]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The ffread and ffwrite functions provide flexible file I/O (FFIO) to record-oriented or byte stream-oriented data in an application-transparent manner.

Function ffweof writes an end-of-file (EOF) if the function is supported for the layer in use.

Function ffweod writes an end-of-data (EOD). If the FFIO specification is stream oriented, this requests truncation of the file at the current position.

Arguments are as follows:

fd Number returned by ffopen(3C).

buf Pointer to the user data buffer.

nb Requested number of bytes to be read or written.

stat Pointer to the returned status structure.

fulp Either FULL or PARTIAL (defined in header file ffio.h); indicates whether I/O should be performed in full or partial record mode.

ubc Unused bit count. The word pointed to by this pointer is in the range of 0 through 7; it is used to specify how many bits in the last byte are not valid data.

The *nb* and *ubc* arguments are used together to determine an exact number of bits to be transferred. *nb* always indicates the number of bytes affected by the data transfer, even if only one bit is transferred in that byte.

If argument *ubc* is omitted, it is assumed to be 0.

If *ubc* is specified, it functions as both an input and output argument. It is passed to ffread or ffwrite as a request to omit processing the specified number of bits in the last byte of the request. For example, to read 2 bits from a file, you should set *nb* to 1, set the word at *ubc* to 6, and call ffread. On completion of the ffread request, the word pointed to by *ubc* is set by ffread to indicate the number of unused bits in the last byte read. As in the preceding example, if only one bit were available to be read, then the word at *ubc* would be set to 7 by ffread.

For ffwrite, *ubc* is an input argument specifying the number of bits in the last byte transferred that should not be written to the file. Function ffwrite normally will not change the value of the word at *ubc*, because, if the specified number of bits is not written, an error is returned.

RETURN VALUES

Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned and, if the *stat* argument is passed, the error value is found in stat.sw_error. If the *stat* parameter is not provided, the error code is found in errno.

If the stat parameter is passed, the sw_stat field is set to one of the following values, indicating the condition that terminated the operation:

Value	Description
FFERR	An error was encountered.
FFEOR	An end-of-record (EOR) was encountered.
FFEOF	An end-of-file (EOF) was encountered.
FFEOD	An end-of-data (EOD) was encountered.
FFCNT	The requested count was satisfied before an EOR, EOF, or EOD was encountered. If the count was satisfied simultaneously with encountering an EOR, the FFEOR status is returned.

The FFSTAT macro, defined in <ffio.h>, can be used to obtain the value of the sw_stat field.

SEE ALSO

ffopen(3C), ffseek(3C)

asgcmd(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

Application Programmer's I/O Guide, publication SG–2168

NAME

ffreada – Provides asynchronous read using flexible file I/O

SYNOPSIS

```
#include <ffio.h>

int ffreada (int fd, char *buf, unsigned nbyte, struct ffsw
*status
[, int fulp [, int *ubc)]];
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The ffreada function is modeled after the reada system call, and the parameters are the same as ffread(3C). The difference between ffread and ffreada is that ffreada performs its work asynchronously whenever possible.

Arguments to ffreada are as follows:

fd Value returned from an ffopen, or ffopens call.

buf Pointer to the buffer in which the data is stored.

nbyte Number of bytes to be written.

status Address of the ffsw status return structure, which is described in the following paragraphs.

fulp Either the value FULL or PARTIAL as defined in header ffio.h. Specifying FULL requests that the file position should be advanced to a point immediately after the next EOR in the file after transferring the requested data. Specifying PARTIAL requests that the file position should be maintained at the bit immediately following the last one transferred. This is an optional argument. If it is not specified, FULL is assumed.

ubc Number of unused bits in the last byte; for example, a single bit can be read by requesting one byte and a *ubc* of 7. This is an optional argument. If it is not specified, 0 is assumed.

Function `ffreada` tries to read *nbyte* bytes from the FFIO file associated with *fd* (opened with *ffopens* or *ffopen*). Data is read into the buffer pointed to by *buf* (see `ffread`(3C)). The function returns control to the user as soon as possible, even when the data cannot be delivered until later. The *stat* parameter is used to notify the caller when the request has completed. The *status* structure has several fields used to determine the status of a given request. The following is a partial description of the status structure:

```
struct ffsw   {
    uint   sw_flag     :1,
           sw_error    :31,
           sw_count    :32,
           sw_stat     :16;
};
```

A call to `ffreada` initiates an asynchronous read of the requested data. You must check for completion of each asynchronous request by using `fffcntl` with the command codes `FC_RECALL` or `FC_ASPOLL`. If such a call to `fffcntl` detects that the request is complete, the status structure is filled in. The *sw_flag* is always set on completion, *sw_error* may contain a system call or library error number (see `intro`(3C)), and *sw_count* contains the number of bytes actually moved. Some FFIO layers with less than byte granularity also return a value in *ubc* that contains the number of bits in the last byte returned that are not valid.

The file position for reading or writing is always the file position at the time of the `ffreada` or `ffwritea`(3C) call. The file's position is incremented at that time by *nbyte* bytes, or the number of bytes remaining in the file, whichever is less. In this way, calls to the `ffreada`, `ffwritea`, and `ffseek`(3C) library functions can be interspersed, and the file position is incremented naturally.

To use asynchronous I/O effectively, all outstanding I/O requests must have their own status structures. Each request must also be checked with an `fffcntl`(3C) call with a command code of `FC_RECALL` or `FC_ASPOLL`. Use code `FC_RECALL` if the caller does not want control returned until the I/O request is complete. Code `FC_ASPOLL` returns control immediately, regardless of the status of the request. This allows the user to check on the completion of a given request without waiting.

RETURN VALUES

If `ffreada` completes successfully, a nonnegative integer is returned. Otherwise, a value of -1 is returned, and `status.sw_error` is set to indicate the error.

ERRORS

The `ffreada` function fails for a variety of reasons, including library and system errors.

EXAMPLES

The following example illustrates different a way to use function `ffreada` to perform a read operation in parallel with other work in a user's process.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <ffio.h>

struct ffsw rdstat, wrstat, dumstat;

main()
{
     char buf[4096*8], buf2[4096*8];
     int fd;

     if ((fd = ffopen("datafile", O_RDONLY)) == -1) {
          perror("open (datafile) failed");
          exit(1);
     }
/*
 *   Assume the file is large.  Read in first 8 sectors, and
 *   simultaneously write out the following 8 sectors.
 */
     ffreada(fd, buf, 4096*8, &rdstat, FULL);
     ffwritea(fd, buf2, 4096*8, &wrstat, FULL);

     /* perform other work here in parallel with I/O completion */

/*
 *   See if either operation is complete with fffcntl(FC_ASPOLL).
 *   Check sw_stat to determine completion.
```

```
 */
     fffcntl(fd, FC_ASPOLL, &rdstat, &dumstat);/* don't wait */
     fffcntl(fd, FC_ASPOLL, &wrstat, &dumstat);/* don't wait */

     if (FFSTAT(rdstat) == 0) reada_is_not_done();
     if (FFSTAT(wrstat) == 0) writea_is_not_done();
/*
 *   wait for read and write to complete.  fffcntl(3) may have to be
 *   called more than once.  Check sw_stat to ensure completion.
 */
     while(FFSTAT(rdstat) == 0)
         fffcntl(fd, FC_RECALL, &rdstat, &dumstat);

     if (FFSTAT(rdstat) == FFERR)
         printf("Read failed, errno=%d", rdstat.sw_error);

     /* wait for write */
     while(FFSTAT(wrstat) == 0)
         fffcntl(fd, FC_RECALL, &wrstat, &dumstat);

     if (FFSTAT(wrstat) == FFERR)
         printf("Read failed, errno=%d", wrstat.sw_error);

     /* input data from ffreada now available in buffer buf */
}
```

SEE ALSO

errno.h(3C), fffcntl(3C), ffopen(3C), ffread(3C), ffseek(3C), ffwritea(3C), intro(3C)

NAME

ffseek, ffbksp – Repositions a flexible file I/O file

SYNOPSIS

```
#include <ffio.h>

#include <unistd.h>

int ffseek (int fd, int pos, int whence [, struct ffsw
*stat]);

int ffbksp (int fd [, struct ffsw *stat]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The ffseek function provides flexible file I/O (FFIO) positioning capability similar to that of lseek(2). In addition to the functionality of lseek, ffseek provides access to limited positioning on blocked files and record-oriented files. Specifying the following:

```
ffseek(fd, 0, 0)
```

always rewinds a file to its initial point, regardless of whether the file is stream, blocked, or tape. When using the syscall layer, this function cannot be used to position a tape. This function allows programs to be written so that their I/O can be controlled by the asgcmd(1) command. Both native and foreign record-oriented I/O, multifile datasets, and performance-oriented layers (such as SDS resident layers and memory-resident layers) are available with this mechanism.

Function ffbksp allows you to perform a backspace operation on those FFIO layers that support it. Currently, this is limited to cos and tape.

Arguments are as follows:

fd Number returned by ffopen(3C).

pos Byte position requested.

whence Specifies one of the following values (defined in header file `stdio.h`):

0 or `SEEK_SET`	Sets the pointer to the value of *pos*. *pos* must be a non-negative integer. (Special case: `ffseek(fd, 0, 0) = rewind`)
`1` or `SEEK_CUR`	Sets the pointer to the current position, plus or minus `*`*arg*. This is supported only in layers that are not record-oriented; layers are specified to `assign -F` as follows: `syscall`, `sds`, `mr` (memory resident), `cache`, `cachea`, `bufa`, `er90`.
`2` or `SEEK_END`	Sets the pointer to the end of the file, minus *pos*. *pos* must be a non-negative integer. Not all layers support this option. (Special case: `ffseek(fd, 0, 2)` = position just in front of the EOD.)

`stat` Pointer to the `ffsw` status return structure.

RETURN VALUES

The return value is the current position in the file after the seek. Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned, and, if the *stat* parameter is passed, the error value is found in `stat.sw_error`. If the *stat* parameter is not provided, the error code is found in `errno`.

EXAMPLES

```
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <foreign.h>

main()
{
      int ret;
      int fd, i, j;

      fd = ffopen("data", O_RDWR | O_CREAT, 0666);

      for (i = 0 ; i < 1000 ; i++)
      {
            ret = ffwrite(fd, &i, sizeof(i));
            if (ret < 0) abort();
      }

      for (i = 0 ; i < 10 ; i++)
      {
            ret = ffseek(fd, i * sizeof(j) * 100, SEEK_SET);
            if (ret < 0) abort();
            ret = ffread(fd, &j, sizeof(j));
            if (ret < 0) abort();
            printf("Value is %d at word %d\n", j, i*100);
      }
}
```

Output from the previous program is as follows:

```
Value is 0 at word 0
Value is 100 at word 100
Value is 200 at word 200
Value is 300 at word 300
Value is 400 at word 400
Value is 500 at word 500
Value is 600 at word 600
Value is 700 at word 700
Value is 800 at word 800
Value is 900 at word 900
```

SEE ALSO

errno.h(3C), ffclose(3C), ffopen(3C), ffread(3C), ffwrite(3C)

NAME

ffwritea – Provides asynchronous write using flexible file I/O

SYNOPSIS

```
#include <ffio.h>

int ffwritea (int fd, char *buf, unsigned nbyte, struct ffsw
*status [,int fulp[, int *ubc]]);
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

CRI extension

DESCRIPTION

The ffwritea function is modeled after the writea system call and the parameters are the same as ffwrite(3C). The difference between ffwrite and ffwritea is that ffwritea performs its work asynchronously when possible.

Arguments to ffwritea are as follows:

fd Value returned from an ffopen, or ffopens call.

buf Pointer to the buffer in which the data is stored.

nbyte Number of bytes to be written.

status Pointer to the ffsw status return structure, described in detail in ffreada(3C).

fulp Either the value FULL or PARTIAL as defined in header foreign.h. FULL requests that an EOR mark be written immediately after the requested data. PARTIAL requests that no such mark be written. This is an optional argument. If it is not specified, FULL is assumed.

ubc Number of unused bits in the last byte; for example, a single bit can be written by requesting 1 byte and a *ubc* of 7. This is an optional argument. If it is not specified, 0 is assumed.

The description of handling asynchronous FFIO is given under `ffreada` and `fflistio`.

ERRORS

Function `ffwritea` fails and the file pointer remains unchanged if any one of a number of conditions occurs. These include many system errors and library errors that result from the operations performed by the FFIO layers performing the I/O.

RETURN VALUES

If `ffwritea` completes successfully, a nonnegative integer is returned. Otherwise, a value of -1 is returned, and `status.sw_error` is set to indicate the error.

EXAMPLES

An example program using `ffwritea` is shown with `ffreada`(3C).

SEE ALSO

`errno.h`(3C), `fffcntl`(3C), `fflistio`(3C), `ffopen`(3C), `ffreada`(3C)

`write`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

ffsetsp – Initiates EOV processing for files opened using flexible file I/O

SYNOPSIS

```
#include <ffio.h>
int ffsetsp (int fd[, struct ffsw *stat]);
```

IMPLEMENTATION

Cray PVP systems

STANDARDS

CRI extension

DESCRIPTION

The ffsetsp function initiates special end-of-volume (EOV) processing for online tape files opened by ffopen or ffopens (see ffopen(3C)), using flexible file I/O (FFIO). Arguments are as follows:

fd Number returned by ffopen or ffopens.

stat Pointer to the status return structure.

This function is valid only for online tape and ER90 files that use the ffio tape layer on the CRAY C90 system.

RETURN VALUES

Function ffsetsp returns 0 if successful. Otherwise, a value of -1 is returned, and, if the stat parameter is passed, the error value is found in stat.sw_error. If the *stat* parameter is not provided, the error code is found in errno.

SEE ALSO

fffcntl(3C), ffopen(3C)

NAME

intro_shmem – Introduction to logically shared memory access routines

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The logically shared, distributed memory access (SHMEM) routines perform various operations on remote and local memory. These routines can minimize overhead and latency and can maximize data bandwidth. Supported operations include remote data transfer, atomic swap, work-shared broadcast and reduction, and barrier synchronization.

The SHMEM routines are described on the following man pages:

- num_pes(3)
- set_d_stream(3)
- shmem_and(3)
- shmem_barrier(3)
- shmem_broadcast(3)
- shmem_cache(3)
- shmem_collect(3)
- shmem_fadd(3)
- shmem_iput(3)
- shmem_ixput(3)
- shmem_lock(3)
- shmem_max(3)
- shmem_min(3)
- shmem_my_pe(3)
- shmem_or(3)
- shmem_prod(3)

- shmem_put(3)
- shmem_quiet(3)
- shmem_stack(3)
- shmem_sum(3)
- shmem_swap(3)
- shmem_wait(3)
- shmem_xor(3)
- start_pes(3)

C, C++, or Fortran data objects are passed by address to SHMEM routines. Target or source arrays that reside on the remote processing element (PE) are typically identified by passing the address of the corresponding data object on the local PE. The existence of a corresponding data object locally implies that a data object is *symmetric* as described on this man page. Most uses of SHMEM routines employ symmetric data objects, but on Cray MPP systems, another class of data object, *asymmetric accessible* data objects, can also be passed to SHMEM routines.

Data objects passed to SHMEM routines can be arrays or scalars, and they fall into the following addressing categories:

Category	**Characteristics**
Symmetric	A remotely accessible data object whose local address maps in a consistent way to a corresponding data object on remote PEs. Remote data objects with this property are accessed through SHMEM routines by using the address of the corresponding data object on the local PE. All symmetric data objects are also *PE-private*. The following data objects are symmetric on Cray MPP systems: • Fortran PE-private data objects in common blocks or with the SAVE attribute • Fortran arrays allocated with shpalloc(3F) • Non-stack C and C++ variables • C and C++ data allocated by shmalloc(3C)

- C and C++ stack variables declared with a `#pragma symmetric` directive
- Fortran stack variables declared with a `CDIR$ SYMMETRIC` directive (CF90 compiler only)

The following data objects are symmetric on Cray PVP systems:

- Fortran arrays and scalars in `TASK COMMON`
- C and C++ scalar and array objects declared in `#pragma taskcommon` directives

Asymmetric accessible — A PE-private data object with a virtual address that can be accessed from other PEs through SHMEM routines. Asymmetric accessible data objects exist on Cray MPP systems only. A data object that is remotely accessible, but not symmetric, can be accessed on Cray MPP systems through SHMEM routines only if the address of the remote data object is communicated between PEs first. The address of the data object on the remote PE must be passed to the SHMEM routines in those cases.

All asymmetric accessible data objects are also PE-private (see below).

The following data objects are asymmetric accessible on Cray MPP systems:

- C and C++ data allocated by `malloc`(3C) and C++ data allocated by the `new` operator,if the same address is a valid heap address on all other PEs
- C and C++ variables with automatic or register storage class, if the private stacks on all other PEs have been extended to the same virtual address
- Fortran arrays allocated with `hpalloc`(3F), if the same address is a valid heap address on all other PEs
- Fortran PE-private data objects on the stack, if the private stacks on all other PEs have been extended to the same virtual address

Remotely accessible — Any symmetric or asymmetric accessible data object.

PE-private — Any data object that is not an MPP Fortran shared array. This includes all C and C++ objects and all Fortran data objects not declared SHARED.

These classifications divide data objects according to the address categories. These classifications apply to all programs except Cray MPP programs linked with the -O SHARED option on mppldr(1). For those programs, only Fortran arrays allocated with shpalloc(3F) and C and C++ data allocated by shmalloc(3C) are symmetric. All other data objects are not symmetric or asymmetric accessible.

SHMEM routines that operate on the same data object on multiple PEs require that symmetric data objects be passed. This restriction is for algorithm simplicity and efficiency. These routines define the set of target PEs by the following triplet of arguments: *PE_start*, *logPE_stride*, and *PE_size*.

Some SHMEM routines, for example, shmem_broadcast(3) and shmem_float_sum_to_all(3), are classified as *collective* routines because they distribute work across a set of PEs. They must be called concurrently by all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* argument triplet.

ROUTINES ON CRAY PVP SYSTEMS

A small subset of the SHMEM routines are available on Cray PVP systems. The IMPLEMENTATION section of each man page indicates if a routine is available on Cray PVP systems. A summary of routines available on Cray PVP systems follows.

The following routines are available from Fortran, C, and C++ on Cray PVP systems:

- barrier(3C)
- barrier(3F)
- shmem_clear_cache_inv(3)
- shmem_get(3)
- shmem_my_pe(3)
- shmem_n_pes(3)
- start_pes(3)
- shmem_put(3)
- shmem_quiet(3)
- shmem_set_cache_inv(3) (unavailable on CRAY T90 systems)
- shmem_set_cache_line_inv(3) (unavailable on CRAY T90 systems)

- shmem_udcflush(3)
- shmem_udcflush_line(3)

The following routine is available from Fortran only (this routine is not intrinsic on Cray PVP systems):

- MY_PE(3I)

The following intrinsic functions are available from C and C++ only (neither of these are intrinsic functions on Cray PVP systems):

- _my_pe(3)
- _num_pes(3)

REDUCTION ROUTINES

The SHMEM reduction routines perform reductions on symmetric arrays. They are described on the following man pages:

- shmem_and(3)
- shmem_max(3)
- shmem_min(3)
- shmem_or(3)
- shmem_prod(3)
- shmem_sum(3)
- shmem_xor(3)

COLLECTIVE ROUTINES

The following man pages describe SHMEM collective routines:

- shmem_and(3)
- shmem_barrier(3)
- shmem_broadcast(3)
- shmem_collect(3)
- shmem_max(3)
- shmem_min(3)
- shmem_or(3)

- shmem_prod(3)
- shmem_sum(3)
- shmem_xor(3)

USING THE *pSync* ARRAY

Multiple *pSync* arrays are often needed if a particular PE calls a SHMEM routine twice. Problems would occur if some PEs in the active set for call 2 arrive at call 2 before processing of call 1 is complete by all PEs in the call 1 active set. Another way to prevent call 2 from beginning before barrier processing at call 1 is complete is to call barrier(3C) or barrier(3F) between call 1 and call 2.

There are two special cases:

- The shmem_barrier(3) routine allows the same *pSync* array to be used on consecutive calls as long as the active PE set does not change.
- If the same collective routine is called multiple times with the same active set, the calls may alternate between two *pSync* arrays. The SHMEM routines guarantee that a first call is completely finished by all PEs by the time processing of a third call begins on any PE.

Because the SHMEM routines restore *pSync* to its original contents, multiple calls that use the same *pSync* array do not require that *pSync* be reinitialized after the first call.

MISCELLANEOUS

Note that Fortran shared variables cannot be passed to any of the SHMEM routines.

Performance of some SHMEM routines is enhanced by using data objects that are cache aligned.

The SHMEM routines reside in libsma.a, which is a library in the default mppldr(1) search path on Cray MPP systems.

The Fortran SHMEM header file for CRAY T3D systems resides in directory /usr/include/mpp on Cray PVP systems. Use the -I /usr/include/mpp option with the f90(1) or cf77(1) commands when compiling Fortran modules that include a SHMEM Fortran header file for CRAY T3D systems.

SEE ALSO

num_pes(3), set_d_stream(3), shmem_and(3), shmem_barrier(3), shmem_broadcast(3), shmem_cache(3), shmem_collect(3), shmem_fadd(3), shmem_iput(3), shmem_ixput(3), shmem_lock(3), shmem_max(3), shmem_min(3), shmem_my_pe(3), shmem_or(3), shmem_prod(3), shmem_put(3), shmem_quiet(3), shmem_stack(3), shmem_sum(3), shmem_swap(3), shmem_wait(3), shmem_xor(3), start_pes(3)

CC(1), cf77(1), f90(1)

mppldr(1)

barrier(3C), shmalloc(3C)

barrier(3F), BARSYNC(3F), hpalloc(3F), shmalloc(3F), shpalloc(3F)

my_pe(3I)

PVM and HeNCE Programmer's Manual, publication SR–2501

Cray MPP Fortran Reference Manual, publication SG–2504

Cray C/C++ Reference Manual, publication SR–2179

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

num_pes – Returns the number of processing elements (PEs) running in an application

SYNOPSIS

C:

```
int  _num_pes(void);
```

Fortran:

```
I  =  NUM_PES()
```

IMPLEMENTATION

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

This function returns the number of PEs running in an application. This is the number of PEs started with the start_pes(3) routine.

SEE ALSO

intro_shmem(3), my_pe(3I)

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_short_and_to_all, shmem_int_and_to_all, shmem_int8_and_to_all, shmem_int4_and_to_all – Performs a logical AND function across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_and_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride, int PE_size,
     short *pWrk, long *pSync);

void shmem_int_and_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride, int PE_size, int
*pWrk,
     long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_AND_TO_ALL(target, source, nreduce,
     PE_start, logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_AND_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see `intro_shmem`(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the result of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* date types are as follows:

Routine	**Data Type**
`shmem_int8_and_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_and_to_all`	Integer, with an element size of 4 bytes

source A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains `max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains `max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the logical AND of the integer variable `FOO` across all even PEs.

```
      INCLUDE "mpp/shmem.fh"

      INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
      DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
      PARAMETER (NR=1)
      REAL PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
      INTEGER FOO, FOOAND
      COMMON /COM/ FOO, FOOAND, PWRK
      INTRINSIC MY_PE

      IF ( MOD(MY_PE(),2) .EQ. 0) THEN
         CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, N$PES/2,
     &       PWRK, PSYNC)
         PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOAND
      ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_and_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_and_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_and_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_and_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_barrier – Performs a barrier operation on a subset of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_barrier(int PE_start, int logPE_stride, int PE_size,
   long *pSync);
```

Fortran:

```
INCLUDE "mpp/shmem.fh"

INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BARRIER_SYNC_SIZE)
CALL SHMEM_BARRIER(PE_start, logPE_stride, PE_size, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

shmem_barrier is a collective synchronization routine. Control returns from shmem_barrier after all PEs in the active set (specified by *PE_start*, *logPE_stride*, and *PE_size*) have called shmem_barrier.

The arguments are as follows:

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_BARRIER_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_BARRIER_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter `shmem_barrier`.

The values of arguments *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same work array must be passed in *pSync* to all PEs in the active set.

`shmem_barrier` ensures that all previously issued local and remote memory stores by any of the PEs in the active set are complete before returning. The data cache is not invalidated, however.

`shmem_barrier` restores the contents of the *pSync* array to the original values on all PEs before any PE returns. This implies that the same *pSync* array may be used on consecutive calls to `shmem_barrier` if the same active PE set is used.

NOTES

The term *symmetric* is defined in `intro_shmem`(3).

`shmem_barrier` resets the values in *pSync* before it returns, so a particular *pSync* buffer needs to be initialized only the first time it is used. If the *pSync* array is initialized at run time, be sure to use some type of synchronization, for example, a call to `barrier`(3F) or `barrier`(3C), before calling `shmem_barrier` the first time.

If the active set does not change, `shmem_barrier` can be called repeatedly with the same *pSync* array. No additional synchronization beyond that implied by `shmem_barrier` itself is necessary in this case.

EXAMPLES

C example:

```
for (i=0; i < _SHMEM_BARRIER_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_barrier(PE_start, logPE_stride, size, pSync);
```

Fortran example:

```
INTEGER PSYNC(SHMEM_BARRIER_SYNC_SIZE)
INTEGER PE_START, LOGPE_STRIDE, PE_SIZE, PSYNC
DATA PSYNC /SHMEM_BARRIER_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_BARRIER(PE_START, LOGPE_STRIDE, PE_SIZE, PSYNC)
```

SEE ALSO

barrier(3C), barrier(3F), intro_shmem(3), set_barrier(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_broadcast, shmem_broadcast32, shmem_broadcast4 – Broadcasts a block of data from one processing element (PE) to one or more target PEs

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_broadcast(void *target, void *source, int nlong, int PE_root,
     int PE_start, int logPE_stride, int PE_size, long *pSync);

void shmem_broadcast32(void *target, void *source, int nlong, int PE_root,
     int PE_start, int logPE_stride, int PE_size, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST(target, source, nlong, PE_root, PE_start, logPE_stride,
     PE_size, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST4(target, source, nlong, PE_root, PE_start,
   logPE_stride, PE_size, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) broadcast routines are collective routines. They copy data object *source* on the processor specified by *PE_root* and store the values at *target* on the other PEs specified by the triplet *PE_start*, *logPE_stride*, *PE_size*. The data is not copied to the target area on the root PE.

The arguments are as follows:

target A symmetric data object with one of the following data types:

Routine	Data Type and Language
`shmem_broadcast`	Any noncharacter type that has an element size of 64 bits. No Fortran derived types or C structures are allowed.
`shmem_broadcast32`	Any noncharacter type that has an element size of 32 bits. No Fortran derived types or C structures are allowed.
`shmem_broadcast4`	Any noncharacter type that has an element size of 32 bits.

source A symmetric data object that can be of any data type that is permissible for the *target* argument.

nlong The number of elements in *source*. For `shmem_broadcast`, this is the number of 64-bit words. For `shmem_broadcast32` and `shmem_broadcast4`, this is the number of 32-bit halfwords. *nlong* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_root Zero-based ordinal of the PE in the active set from which the data is copied. Must be greater than or equal to 0 and less than `PE_size`. *PE_root* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *log_PE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_BCAST_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_BCAST_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter `shmem_barrier`().

The values of arguments *PE_root*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* data objects and the same *pSync* work array must be passed to all PEs in the active set.

Upon return from a broadcast routine, the following are true for the local PE:

- If the current PE is not the root PE, the *target* data object is updated.
- The data cache region that is mapped to the *target* data object is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM broadcast routines restore *pSync* to its original contents. Multiple calls to SHMEM routines that use the same *pSync* array do not require that *pSync* be reinitialized after the first call.

You must ensure the that the *pSync* array is not being updated by any PE in the active set while any of the PEs participates in processing of a SHMEM broadcast routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.

- A *pSync* array may be reused on a subsequent SHMEM broadcast routine only if none of the PEs in the active set are still processing a prior SHMEM broadcast routine call that used the same *pSync* array. In general, this can be ensured only by doing some type of synchronization. However, in the special case of SHMEM routines being called with the same active set, you can allocate two *pSync* arrays and alternate between them on successive calls.

EXAMPLES

C example:

```
for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_broadcast(target, source, nlong, PE_root, PE_start,
  logPE_stride, PE_size, pSync);
```

Fortran example:

```
      INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
      INTEGER TARGET, SOURCE, NLONG, PE_ROOT, PE_START,
&       LOGPE_STRIDE, PE_SIZE, PSYNC
      COMMON /COM/ TARGET, SOURCE

      DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/

      CALL SHMEM_BROADCAST(TARGET, SOURCE, NLONG, PE_ROOT, PE_START,
&       LOGPE_STRIDE, PE_SIZE, PSYNC)
```

SEE ALSO

intro_blacs(3S), intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_set_cache_inv, shmem_set_cache_line_inv, shmem_clear_cache_inv, shmem_udcflush, shmem_udcflush_line – Controls data cache utilities

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_set_cache_inv(void);

void shmem_set_cache_line_inv(long *target);

void shmem_clear_cache_inv(void);

void shmem_udcflush(void);

void shmem_udcflush_line(long *target);
```

Fortran:

```
INCLUDE "mpp/shmem.fh"

CALL SHMEM_SET_CACHE_INV

CALL SHMEM_SET_CACHE_LINE_INV(target)

CALL SHMEM_CLEAR_CACHE_INV

CALL SHMEM_UDCFLUSH

CALL SHMEM_UDCFLUSH_LINE(target)
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

shmem_set_cache_inv and shmem_set_cache_line_inv are not supported on CRAY T90 systems

DESCRIPTION

The following argument is passed to the cache line control routines:

target A data object that is local to the processing element (PE). *target* can be of any noncharacter type. If you are using the CF90 compiler, it can be of any kind.

`shmem_set_cache_inv` enables automatic cache coherency mode. When this mode is enabled on CRAY T3D systems, remote writes into the calling PE's local memory from other PEs via `shmem_put`(3) or `shmem_swap`(3) do not disrupt data cache coherency because the affected cache line in the local PE gets invalidated automatically.

`shmem_set_cache_line_inv` enables automatic cache coherency mode for the cache line associated with the address of *target* only.

`shmem_set_cache_inv` and `shmem_set_cache_line_inv` are no-ops on CRAY Y-MP, CRAY J90, CRAY C90, and CRAY T3E systems because no data cache exists or because hardware or the operating system ensures cache coherency.

`shmem_set_cache_inv` and `shmem_set_cache_line_inv` are not provided on CRAY T90 systems because hardware automatic cache coherency mode is unavailable.

`shmem_clear_cache_inv` disables automatic cache coherency mode previously enabled by `shmem_set_cache_inv` or `shmem_set_cache_line_inv`.

On CRAY Y-MP, CRAY J90, CRAY C90, and CRAY T3E systems, automatic cache coherency cannot be disabled, so `shmem_clear_cache_inv` is a no-op. On CRAY T90 systems, automatic cache coherency mode is unavailable, so `shmem_clear_cache_inv` is also a no-op.

`shmem_udcflush` makes the entire user data cache coherent.

On CRAY Y-MP, CRAY J90, CRAY C90, and CRAY T3E systems, `shmem_udcflush` and `shmem_udcflush_line` are no-ops because no data cache exists or because hardware or the operating system ensures cache coherency.

On CRAY T3D and CRAY T90 systems, `shmem_udcflush` makes the data cache coherent by invalidating it. This has the same effect as the `#pragma _CRI suppress` directive (in C) and the `CDIR$ SUPPRESS` directive (in Fortran).

`shmem_udcflush_line` makes coherent the cache line that corresponds with the address specified by *target*.

On CRAY Y-MP, CRAY J90, CRAY C90, and CRAY T3E systems, `shmem_udcflush_line` is a no-op because no data cache exists or because hardware or the operating system ensures cache coherency.

On CRAY T3D systems, `shmem_udcflush_line` makes the data cache line coherent by invalidating it. On CRAY T90 systems, `shmem_udcflush_line` makes the data cache line coherent by invalidating the entire data cache. This has the same effect as the `#pragma _CRI suppress` directive (in C) and the `CDIR$ SUPPRESS` directive (in Fortran) when these directives specify a variable name.

NOTES

On CRAY T3D systems, automatic cache invalidation is disabled prior to any user call to `shmem_set_cache_inv` or `shmem_set_cache_line_inv`.

Because hardware or the operating system ensures that the cache never becomes incoherent on CRAY T3E, CRAY Y-MP, CRAY J90, or CRAY C90 systems, all cache-management routines are no-ops on these architectures.

SEE ALSO

`intro_shmem`(3), `shmem_put`(3), `shmem_swap`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_collect, shmem_fcollect, shmem_collect32, shmem_fcollect32, shmem_collect4, shmem_fcollect4 – Concatenates blocks of data from multiple processing elements (PEs) to an array in every PE

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_collect(void *target, void *source, int nlong, int PE_start,
   int logPE_stride, int PE_size, long *pSync);

void shmem_fcollect(void *target, void *source, int nlong, int PE_start,
   int logPE_stride, int PE_size, long *pSync);

void shmem_collect32(void *target, void *source, int nlong, int PE_start,
   int logPE_stride, int PE_size, long *pSync);

void shmem_fcollect32(void *target, void *source, int nlong, int PE_start,
   int logPE_stride, int PE_size, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_COLLECT(target, source, nlong, PE_start, logPE_stride, PE_size,
   pSync)

CALL SHMEM_FCOLLECT(target, source, nlong, PE_start, logPE_stride, PE_size,
   pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER nlong
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_COLLECT4(target, source, nlong, PE_start, logPE_stride, PE_size,
   pSync)

CALL SHMEM_FCOLLECT4(target, source, nlong, PE_start, logPE_stride, PE_size,
   pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) collective routines concatenate *nlong* 64- or 32-bit data items from the source array into the target array, over the set of PEs defined by *PE_start*, *log2PE_stride*, and *PE_size*, in processor number order. The resultant target array contains the contribution from PE *PE_start* first, then the contribution from PE *PE_start* + *PE_stride* second, and so on. The collected result is written to the target array for all PEs in the active set.

The resulting target array is as follows:

```
-----------------------------------------------------------
   source(1..nlong)
       from PE (PE_start + 0 * (2**logPE_stride))
-----------------------------------------------------------
   source(1..nlong)
       from PE (PE_start + 1 * (2**logPE_stride))
-----------------------------------------------------------
   ...
-----------------------------------------------------------
   source(1..nlong)    from
       PE (PE_start + (PE_size - 1) * (2**logPE_stride))
-----------------------------------------------------------
```

The arguments are as follows:

target A symmetric array. The *target* argument must be large enough to accept the concatenation of the source arrays on all PEs. The data types are as follows:

Routine	Data Type
`shmem_collect`, `shmem_fcollect`	Any data type with an element size of 64 bits. Fortran derived types, Fortran character type, and C structures are not permitted.
`shmem_collect32`, `shmem_fcollect32`, `shmem_collect4`, `shmem_fcollect4`	Any data type with an element size of 32 bits. Fortran derived types, Fortran character type, and C structures are not permitted.

source A symmetric data object that can be of any type permissible for the *target* argument.

nlong The number of elements in the source array. *nlong* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

The *nlong* argument must be equal on all PEs for `shmem_fcollect` and `shmem_fcollect32`. The *nlong* argument can be different across PEs for `shmem_collect` and `shmem_collect32`.

PE_start The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pSync A symmetric work array.

In C, *pSync* must be of type `int` and size `_SHMEM_COLLECT_SYNC_SIZE`.

In Fortran, *pSync* must be of type integer and size `SHMEM_COLLECT_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` in C or `SHMEM_SYNC_VALUE` in Fortran before any of the PEs in the active set enter `shmem_barrier`().

The values of arguments *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same target and source arrays and the same *pSync* work array must be passed to all PEs in the active set.

Upon return from a collective routine, the following are true for the local PE:

- The target array is updated.
- The data cache region that is mapped to the target data object is coherent in all receiving PEs.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM collective routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM collective routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* array can be reused on a subsequent SHMEM collective routine only if none of the PEs in the active set are still processing a prior SHMEM collective routine call that used the same *pSync* array. In general, this may be ensured only by doing some type of synchronization. However, in the special case of SHMEM routines being called with the same active set, you can allocate two *pSync* arrays and alternate between them on successive calls.

Routine shmem_collect allows the value of *nlong* to be different on each participating processor. Routine shmem_fcollect, however, expects the value of *nlong* to be the same for all processors. Because the value for *nlong* is the same on all processors for calls to shmem_fcollect(), the speed of this routine is greatly improved over routine shmem_collect(), due to the reduced overhead of calculating where each contribution belongs in the target array.

The collective routines operate on active PE sets that have a non-power-of-two *PE_size* with some performance degradation. They operate with no performance degradation when *nlong* is a non-power-of-two value.

EXAMPLES

Example in C:

```
for (i=0; i < _SHMEM_COLLECT_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
}
barrier(); /* Wait for all PEs to initialize pSync */
shmem_collect(target, source, 64, pe_start, logPE_stride,
   pe_size, pSync);
```

Example in Fortran:

```
INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_COLLECT(TARGET, SOURCE, NLONG, ROOT, PE_START,
&  LOGPE_STRIDE, PE_SIZE, PSYNC)
CALL SHMEM_COLLECT(TARGET, SOURCE, 64, PE_START, LOGPE_STRIDE,
&  PE_SIZE, PSYNC)
```

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_fence – assures ordering of delivery of PUTs

SYNOPSIS

C:

```
#include <mpp/shmem.h>
void shmem_fence(void);
```

Fortran:

```
CALL SHMEM_FENCE
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit on Cray PVP systems

DESCRIPTION

The shmem_fence function ensures ordering of PUT (remote write) operations. All PUT operations issued to a particular processing element (PE) prior to the call to shmem_fence are guaranteed to be delivered before any PUT operations to the same processing element (PE) which follow the call to shmem_fence.

NOTES

The shmem_quiet function should be called if ordering of PUTs is desired when multiple remote PEs are involved.

The shmem_fence function is provided in C in the form of a macro.

The shmem_fence function is a no-op on CRAY T3D systems.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_finc, SHMEM_INT4_FINC – Performs a fetch-and-increment operation on a remote data object

SYNOPSIS

C or C++:

```
#include <mpp/shmem.h>

short shmem_short_finc(short *target, int pe);
```

Fortran:

```
INTEGER pe

INTEGER*4 SHMEM_INT4_FINC
ires = SHMEM_INT4_FINC(target, pe)
```

IMPLEMENTATION

CRAY T3E systems

DESCRIPTION

shmem_finc performs a fetch-and-increment. It increments the *target* on processing element (PE) *pe* by one and returns the previous contents of *target* as an atomic operation.

The arguments are as follows:

target The remotely accessible integer data object to be updated on the remote PE. If you are using C, the type of *target* should match that implied in the SYNOPSIS section. If you are using the CF90 compiler, it must be of the following type:

SHMEM_INT4_FINC	4-byte INTEGER

pe An integer that indicates the PE number on which *target* is to be updated. If you are using the CF90 compiler, it must be a default integer value.

NOTES

The term remotely accessible is defined in intro_shmem(3).

The fetch-and-increment routines write directly into another PE's memory without notifying that PE, so the receiving PE should normally set automatic cache invalidation by use of shmem_set_cache_inv(3) or shmem_set_cache_line_inv(3). If this is not done, the receiving PE's data cache might become incoherent.

You can use the fetch-and-increment routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the target addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using shmem_udcflush(), shmem_udcflush_line(), the #pragma _CRI suppress directive (from C), or the CDIR$ SUPPRESS directive (from Fortran).

RETURN VALUES

The contents that had been at the target address on the remote PE prior to the increment.

SEE ALSO

intro_shmem(3), shmem_clear_cache_inv(3), shmem_set_cache_inv(3), shmem_set_cache_line_inv(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

NAME

shmem_iget, shmem_iput, shmem_iget32, shmem_iput32, shmem_iget4, shmem_iput4 – Transfers strided data between remote and local memory

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_iget(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iput(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iget32(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);

void shmem_iput32(void *target, void *source, int target_inc,
   int source_inc, int len, int pe);
```

Fortran (CF77 and CF90):

```
INTEGER target_inc, source_inc, len, pe
CALL SHMEM_IGET(target, source, target_inc, source_inc, len, pe);

CALL SHMEM_IPUT(target, source, target_inc, source_inc, len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER target_inc, source_inc, len, pe
CALL SHMEM_IGET4(target, source, target_inc, source_inc, len, pe);

CALL SHMEM_IPUT4(target, source, target_inc, source_inc, len, pe);
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The `iget` and `iput` routines transfer strided arrays between local and remote memory.

The `shmem_iget`, `shmem_iget32`, and `shmem_iget4` routines transfer data from the remote address defined by *pe* and *source* into the local address defined by *target*. The function returns when the data has been loaded into the local data array. On CRAY T3D systems, this function uses remote loads.

The `shmem_iput`, `shmem_iput32`, and `shmem_iput4` routines transfer data from the local address defined by *source* into the remote address defined by *pe* and *target*. The function returns when the data has been sent from the chip (but it may not yet have arrived at the other processing element (PE)). On CRAY T3D systems, this function uses remote stores.

The arguments are as follows:

target Array to be updated. For `iget` routines, *target* resides on the local PE and need not be remotely accessible. For `iput` routines, *target* resides on the remote PE and must be remotely accessible.

The data types are as follows:

Routine	Data Type
`shmem_iget`, `shmem_iput`	Any data type with an element size of 64 bits. Fortran derived types, Fortran character type, and C structures are not permitted.
`shmem_iget4`, `shmem_iput4`, `shmem_iget32`, `shmem_iput32`	Any data type with an element size of 32 bits. Fortran derived types, Fortran character type, and C structures are not permitted.

source Array to be transferred. This must be a remotely accessible data object. The *source* argument can have any of the data types permitted for *target*.

target_inc The stride between consecutive elements of the *target* array. The stride is scaled by the element size of the *target* array. A value of 1 indicates contiguous data. *target_inc* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

source_inc The stride between consecutive elements of the source array. The stride is scaled by the element size of the *source* array. A value of 1 indicates contiguous data. *source_inc* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

len Number of strided elements that will be transferred from the source to the *target* array. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

EXAMPLES

The `shmem_iget` routine is semantically equivalent to the following:

C example:

```
for ( i=0; i<len; i++ ) {
        shmem_get( target, source, 1, pe );
        source += source_inc;
        target += target_inc;
}
```

Fortran example:

```
SRC_IDX = 1
TAR_IDX = 1
DO I = 1, LEN
        CALL SHMEM_GET(TARGET(TAR_IDX), SOURCE(SRC_IDX), 1, PE)
        SRC_IDX = SRC_IDX + SOURCE_INC
        TAR_IDX = TAR_IDX + TARGET_INC
ENDDO
```

NOTES

On CRAY T3D systems, you can use the `put` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

See `intro_shmem`(3) and `shmem_put`(3) for more information about remotely accessible data objects.

SEE ALSO

`intro_shmem`(3), `shmem_put`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_ixget, shmem_ixput, shmem_ixget32, shmem_ixput32, shmem_ixget4, shmem_ixput4 – Transfers strided data between remote and local memory

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_ixget(void *target, void *source, long *source_index,
   int len, int pe);

void shmem_ixput(void *target, void *source, long *target_index,
   int len, int pe);

void shmem_ixget32(void *target, void *source, short *source_index,
   int len, int pe);

void shmem_ixput32(void *target, void *source, short *target_index,
   int len, int pe);
```

Fortran (CF77 and CF90):

```
INTEGER len, pe
CALL SHMEM_IXGET(target, source, source_index, len, pe);

CALL SHMEM_IXPUT(target, source, target_index(*), len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER len, pe
CALL SHMEM_IXGET4(target, source, source_index, len, pe);

CALL SHMEM_IXPUT4(target, source, target_index, len, pe);
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The `ixget` and `ixput` routines gather or scatter data between remote and local memory.

`shmem_ixget`, `shmem_ixget32`, and `shmem_ixget4` transfer data from the remote indexed array defined by *pe* and *source* into the local array defined by *target*. The function returns when the data has been loaded into the local data array. On CRAY T3D systems, this function uses remote loads.

`shmem_ixput`, `shmem_ixput32`, and `shmem_ixput4` transfer data from the local address defined by *source* into the remote indexed address defined by *pe* and *target*. The function returns when the data has been sent from the chip (but it may not yet have arrived at the other processing element (PE)). On CRAY T3D systems, this function uses remote stores.

The arguments are as follows:

target Array to be updated. For `ixget` routines, *target* resides on the local PE and need not be remotely accessible. For `ixput` routines, *target* resides on the remote PE and must be remotely accessible.

The data type of *target* varies with the routine being called. When calling from C, see the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_ixget`, `shmem_ixput`	Any noncharacter date type with a storage size of 8 bytes per element.
`shmem_ixget32`, `shmem_ixput32`, `shmem_ixget4`, `shmem_ixput4`	Any noncharacter data type with a storage size of 4 bytes per element.

source
A remotely accessible data object that can be of any data type permissible for the *target* argument.

target_index
Used on the *ixput* routines. Array of zero-based indexes into the target array. In C, *target_index* is the type implied in the SYNOPSIS section. In Fortran, the type of *target_index* is as follows:

Routine	Data Type
`shmem_ixput`	Integer, with a storage size of 8 bytes per element.
`shmem_ixput4`	Integer, with a storage size of 4 bytes per element.

source_index Used on the `ixget` routines. Array of zero-based indexes into the source array. In C, *target_index* is the type implied in the SYNOPSIS section. In Fortran, the type of *target_index* is as follows:

Routine	Data Type
`shmem_ixget`	Integer, with a storage size of 8 bytes per element.
`shmem_ixget4`	Integer, with a storage size of 4 bytes per element.

len Number of elements to transfer. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

NOTES

On CRAY T3D systems, `shmem_ixget` uses the prefetch queue to gather the remote data. Using the prefetch queue gives consistently good performance results independent of the size of the source array or of the distribution of values in the `source_index` array.

On CRAY T3D systems, you can use the `ixput` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

See intro_shmem(3) and shmem_put(3) for more information about remotely accessible data objects.

EXAMPLES

The shmem_ixget routine is semantically equivalent to the following:

C:

```
for ( i=0; i<len; i++ ) {
   shmem_get( &(target[i]), source + source_index[i], 1, pe );
}
```

Fortran:

```
DO I = 1, LEN
    CALL SHMEM_GET( TARGET(I), SOURCE(SOURCE_INDEX(I) + 1), 1, PE )
ENDDO
```

SEE ALSO

intro_shmem(3), shmem_put(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_clear_lock, shmem_set_lock, shmem_test_lock – Releases, locks, and tests a mutual exclusion memory lock

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_clear_lock(long *lock);

void shmem_set_lock(long *lock);

int shmem_test_lock(long *lock);
```

Fortran (CF77 and CF90):

```
INTEGER lock, SHMEM_TEST_LOCK

CALL SHMEM_CLEAR_LOCK(lock)

CALL SHMEM_SET_LOCK(lock)

I = SHMEM_TEST_LOCK(lock)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shmem_clear_lock routine releases a lock previously set by shmem_set_lock after ensuring that all local and remote stores initiated in the critical region are complete.

The shmem_set_lock routine sets a mutual exclusion lock after waiting for the lock to be freed by any other task currently holding the lock. Waiting tasks are assured of getting the lock in a first-come, first-served manner.

The shmem_test_lock function sets a mutual exclusion lock only if it is currently cleared. By using this function, a task can avoid blocking on a set lock. If the lock is currently set, the function returns without waiting.

These routines are appropriate for protecting a critical region from simultaneous update by multiple tasks. They accept the following argument:

lock A symmetric data object that is a scalar value or an array of length 1. This data object must be set to 0 on all processing elements (PEs) prior to the first use. *lock* must be of type integer. If you are using the CF90 compiler, it must be of default kind.

NOTES

The term *symmetric data object* is defined on `intro_shmem`(3).

The `shmem_clear_lock` and `shmem_set_lock` routines do not accept CRAFT shared variables. Use the `CLEAR_LOCK`(3F) and `SET_LOCK`(3F) routines if you want to use CRAFT shared locks. For more information on CRAFT, see the *Cray MPP Fortran Reference Manual*, publication SR–2504.

RETURN VALUES

The `shmem_test_lock` function returns 0 if the lock was originally cleared and this call was able to set the lock. A value of 1 is returned if the lock had been set and the call returned without waiting to set the lock.

SEE ALSO

`intro_shmem`(3)

`CLEAR_LOCK`(3F), `SET_LOCK`(3F), `TEST_LOCK`(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

NAME

shmem_short_max_to_all, shmem_int_max_to_all, shmem_int8_max_to_all, shmem_comp8_max_to_all, shmem_int4_max_to_all, shmem_real4_max_to_all – Performs a maximum function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_max_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *pWrk, long *pSync);

void shmem_int_max_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_MAX_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_COMP8_MAX_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_MAX_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL4_MAX_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_max_to_all`	Complex, with an element size equal to two 8-byte real values
`shmem_int8_max_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_max_to_all`	Integer, with an element size of 4 bytes
`shmem_real4_max_to_all`	Real, with an element size of 4 bytes

source
A symmetric array of length *nreduce* elements that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the maximum value of real variable FOO across all even PEs.

```
INCLUDE "mpp/shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMAX, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMAX, PWRK
INTRINSIC MY_PE

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
   CALL SHMEM_REAL8_MAX_TO_ALL(FOOMAX, FOO, NR, 0, 1, N$PES/2,
&       PWRK, PSYNC)
   PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOMAX
ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_max_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_max_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_max_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_max_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_min_to_all, shmem_int_min_to_all, shmem_float_min_to_all, shmem_double_min_to_all, shmem_int8_min_to_all, shmem_real8_min_to_all, shmem_int4_min_to_all, shmem_real4_min_to_all – Performs a minimum function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_min_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *pWrk, long *pSync);

void shmem_int_min_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);

void shmem_float_min_to_all(float *target, float *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, float *pWrk, long *pSync);

void shmem_double_min_to_all(double *target, double *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, double *pWrk, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_MIN_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL8_MIN_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_MIN_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL4_MIN_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_int8_min_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_min_to_all`	Integer, with an element size of 4 bytes
`shmem_real8_min_to_all`	Real, with an element size of 8 bytes
`shmem_real4_min_to_all`	Real, with an element size of 4 bytes

source
A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the minimum value of real variable `FOO` across all the even PEs.

```
INCLUDE "mpp/shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMIN, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMIN, PWRK
INTRINSIC MY_PE

IF ( MOD(MY_PE(),2) .EQ. 0) THEN
   CALL SHMEM_REAL8_MIN_TO_ALL(FOOMIN, FOO, NR, 0, 1, N$PES/2,
&       PWRK, PSYNC)
   PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOMIN
ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_min_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_min_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_min_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_min_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_my_pe, shmem_n_pes – Returns processing element (PE) information

SYNOPSIS

C:

```
int  shmem_my_pe(void);
int  shmem_n_pes(void);
```

Fortran (CF77 and CF90):

```
INTEGER  SHMEM_MY_PE,  SHMEM_N_PES
MYPE  =  SHMEM_MY_PE()
NPES  =  SHMEM_N_PES()
```

IMPLEMENTATION

Cray MPP systems

libsma in Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The shmem_my_pe function returns the number of the local PE.

The shmem_n_pes function returns the total number of PEs in the current application.

NOTES

The Cray Research C and CF77 compilers provide the _my_pe() and MY_PE() intrinsic functions on Cray MPP systems, which generate fast inline code to return the current PE. The shmem_my_pe function is most useful on Cray PVP systems and when using the CF90 compiler because no such intrinsic is available.

The Cray Research C and CF77 compilers provide the _num_pes() intrinsic function on Cray MPP systems and N$PES constant, which generate fast inline code to return the number of PEs. The shmem_n_pes function is most useful when using the CF90 compiler because no such intrinsic or constant is available.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, SR–2196

NAME

shmem_short_or_to_all, shmem_int_or_to_all, shmem_int8_or_to_all, shmem_int4_or_to_all – Performs a logical OR function reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_or_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *pWrk, long *pSync);

void shmem_int_or_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_OR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_OR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see `intro_shmem`(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	**Data Type**
`shmem_int8_or_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_or_to_all`	Integer, with an element size of 4 bytes

source
A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent in all receiving PEs.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the logical OR of the integer variable `FOO` across all even PEs.

```
      INCLUDE "mpp/shmem.fh"

      INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
      DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
      PARAMETER (NR=1)
      REAL PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
      INTEGER FOO, FOOOR
      COMMON /COM/ FOO, FOOOR, PWRK
      INTRINSIC MY_PE

      IF ( MOD(MY_PE(),2) .EQ. 0) THEN
         CALL SHMEM_INT8_OR_TO_ALL(FOOOR, FOO, NR, 0, 1, N$PES/2,
     &       PWRK, PSYNC)
         PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOOR
      ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_or_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_or_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_or_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_or_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_prod_to_all, shmem_int_prod_to_all, shmem_float_prod_to_all, shmem_double_prod_to_all, shmem_complexf_prod_to_all, shmem_complexd_prod_to_all, shmem_int8_prod_to_all, shmem_real8_prod_to_all, shmem_comp8_prod_to_all, shmem_int4_prod_to_all, shmem_real4_prod_to_all, shmem_comp4_prod_to_all – Performs a product reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_prod_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *pWrk, long *pSync);

void shmem_int_prod_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);

void shmem_float_prod_to_all(float *target, float *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, float *pWrk, long *pSync);

void shmem_double_prod_to_all(double *target, double *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, double *pWrk, long *pSync);

void shmem_complexf_prod_to_all(float complex *target,
   float complex *source, int nreduce, int PE_start,
   int logPE_stride, int PE_size, float complex *pWrk,
   long *pSync);

void shmem_complexd_prod_to_all(double complex *target,
   double complex *source, int nreduce, int PE_start,
   int logPE_stride, int PE_size, double complex *pWrk,
   long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL8_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_COMP8_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL4_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_COMP4_PROD_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_prod_to_all`	Complex, with an element size equal to two 8-byte real values
`shmem_comp4_prod_to_all`	Complex, with an element size equal to two 4-byte real values
`shmem_int8_prod_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_prod_to_all`	Integer, with an element size of 4 bytes
`shmem_real8_prod_to_all`	Real, with an element size of 8 bytes
`shmem_real4_prod_to_all`	Real, with an element size of 4 bytes

source
A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the product of the real variable `FOO` across all the even PEs.

```
 INCLUDE "mpp/shmem.fh"

 INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
 DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
 PARAMETER (NR=1)
 REAL FOO, FOOPROD, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
 COMMON /COM/ FOO, FOOPROD, PWRK
 INTRINSIC MY_PE

 IF ( MOD(MY_PE(),2) .EQ. 0) THEN
    CALL SHMEM_COMP8_PROD_TO_ALL(FOOPROD, FOO, NR, 0, 1, N$PES/2,
&       PWRK, PSYNC)
    PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOPROD
 ENDIF
```

Example 2: Consider the following C call:

```
shmem_short_prod_to_all(target, source, 3, 0, 0, 8, pwrk, psync);
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_short_prod_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_short_prod_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_short_prod_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_get, shmem_put, shmem_get32, shmem_put32, shmem_get4, shmem_put4 – Transfers data between local and remote memory

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_get(void *target, void *source, int len,
   int pe);

void shmem_put(void *target, void *source, int len,
   int pe);

void shmem_get32(void *target, void *source, int len,
   int pe);

void shmem_put32(void *target, void *source, int len,
   int pe);
```

Fortran (CF77 and CF90):

```
INTEGER len, pe
CALL SHMEM_GET(target, source, len, pe);

CALL SHMEM_PUT(target, source, len, pe);
```

Fortran 90 (CF90) only:

```
INTEGER len, pe
CALL SHMEM_GET4(target, source, len, pe);

CALL SHMEM_PUT4(target, source, len, pe);
```

IMPLEMENTATION

Cray MPP systems

shmem_get and shmem_put are included in the Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The put and get routines provide a high-performance method for transferring data between remote and local memory, but they require careful use of variables and addresses.

shmem_get, shmem_get32, and shmem_get4 transfer data from the remote data object defined by *pe* and *source* into the local data object defined by *target*. The function returns when the data has been stored in the local data object.

shmem_put, shmem_put32, and shmem_put4 transfer data from the local data object defined by *source* into the remote data object defined by *pe* and *target*. The function returns when the data has been sent from the local processing element (PE), but it may not yet have arrived at the other PE.

The arguments are as follows:

target Array to be updated. For get routines, *target* resides on the local PE. For put routines, *target* resides on the remote PE and must be remotely accessible.

The data type of *target* varies with the version of the put routine being called and the language used. The *target* data types are as follows:

Routine	Data Type
shmem_get, shmem_put	Any noncharacter type that has a storage size equal to 8 bytes.
shmem_get32, shmem_put32, shmem_get4, shmem_put4	Any noncharacter type that has a storage size equal to 4 bytes.

source
Array to be copied. The *source* argument can have any data type permitted for *target*.

For put routines, *source* resides on the local PE. For get routines, *source* resides on the remote PE and must be remotely accessible.

len
Number of elements in the *target* and *source* arrays. *len* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pe
PE number of the remote PE. *pe* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

For the most part, shmem_get32() and shmem_put32() transfer data at the same rate as their get and put counterparts. However, performance for the 32-bit routines degrades significantly if one of the target or source arrays is aligned in memory on a 64-bit word boundary and the other is not. You can avoid this problem by cache aligning the source and target arrays and then always beginning transfers with either even-numbered elements of both arrays or odd-numbered elements of both arrays. You should never begin transfers with one even- and one odd-numbered element. The following C example shows this:

```
#pragma _CRI cache_align target, source
short target[256], source[256];

example()
{
        shmem_put32(target, source, nlong, pe);
            /* runs fast   */
        shmem_put32(&target[5], &source[15], nlong, pe);
            /* runs fast   */
        shmem_put32(target, &source[5], nlong, pe);
            /* runs slower */
}
```

EXAMPLES

This simple shmem_put example runs on Cray MPP systems and Cray PVP systems.

```
#include <mpp/shmem.h>

main()
{

        long source[10] = { 1, 2, 3, 4, 5,
                            6, 7, 8, 9, 10 };

#if     _CRAY1
#pragma taskcommon target
#endif
        static long target[10];
```

```
#if     _CRAY1
        start_pes(2);
#endif

        if (_my_pe() == 0) {
                /* put 10 words into target on PE 1 */
                shmem_put(target, source, 10, 1);
        }

        barrier();                          /* wait for completion */

        if (_my_pe() == 1) {
                shmem_udcflush();
        }

        printf("target[0] on PE %d is %d0, _my_pe(), target[0]);


}
```

NOTES

On CRAY T3D systems, you can use the `put` routines without setting the remote write invalidate bit if you know, prior to the call, that the data at the *target* addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using `shmem_udcflush`(), `shmem_udcflush_line`(), the `#pragma _CRI suppress` directive (from C), or the `CDIR$ SUPPRESS` directive (from Fortran).

CRAY J90, CRAY Y-MP, and CRAY C90 systems do not use scalar caches for multitasked programs, so calls to cache management functions are not needed when using the `shmem_put` function.

CRAY T3E systems ensure that data caches stay coherent while doing remote data writes, so calls to cache management functions are not needed when using the `put` routines.

See intro_shmem(3) for a definition of the term *remotely accessible*. Note that remote data objects must be declared on a TASKCOMMON compiler directive on Cray PVP systems.

On Cray MPP systems, use dynamically allocated data objects with the put and get routines carefully because:

- Each PE may have a different address for a stack or heap data object.
- These data objects are not remotely accessible in some CRAFT programs.

Stack variables are defined only for the lifetime of the function that declares them, and many variables (located in many functions) can share the same address. For dynamically allocated data, you cannot simply specify a local variable name for the remote address. Instead, you must explicitly transfer the remote address to the PE calling one of these routines. You must also ensure, through your algorithm, that the address remains valid as long as the other PE may be using it. You risk receiving operand range errors if the remote address is not accessible remotely.

These routines perform no synchronization. You must be sure that both PEs are ready for the action to take place. If you use shmem_get, for example, you must be sure that the data you are copying from the other PE has been updated or you will pick up old data or partially updated data.

On CRAY T3D systems, shmem_put transmits data with about twice the bandwidth of shmem_get. Algorithms get higher performance if shmem_put is used on the sending processor instead of using shmem_get on the receiving processor.

SEE ALSO

intro_shmem(3), shmem_iput(3), shmem_ixput(3), shmem_quiet(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_quiet – Waits for completion of all outstanding remote writes issued by a processing element (PE)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_quiet(void);
```

Fortran:

```
CALL SHMEM_QUIET
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

shmem_quiet waits for completion of all outstanding remote writes initiated from the current PE. Remote writes are issued by calls to shmem_put and related put routines. When control returns from shmem_put, the data is delivered to the communication circuitry but has not yet arrived at the remote PE. shmem_quiet does not return until all data is delivered to the remote PE's memory.

NOTES

shmem_quiet is an inlined macro in C. To get the function form, the shmem_quiet macro name must be undefined by use of the #undef shmem_quiet cpp(1) directive.

shmem_quiet is most useful as a way of ensuring ordering of delivery of several put operations. For example, you might use shmem_quiet to await delivery of a block of data before issuing another put, which sets a completion flag in the remote PE.

shmem_quiet is not usually needed if barrier(3C), barrier(3F), BARSYNC(3F), or shmem_barrier(3) are called. The barrier routines all wait for the completion of outstanding remote writes (puts).

EXAMPLES

```
PROGRAM COMPFLAG
INTEGER FLAG_VAR, ARRAY(100), RECEIVER, SENDER
COMMON/FLAG/FLAG_VAR
COMMON/DATA/ARRAY
INTRINSIC MY_PE

FLAG_VAR = 0
CALL BARRIER                    ! wait for FLAG_VAR to be initialized

SENDER = 0                      ! PE 0 sends the data
RECEIVER = 1                    ! PE 1 receives the data
IF (MY_PE() .EQ. 0) THEN
  ARRAY = 33
  CALL SHMEM_PUT(ARRAY, ARRAY, 100, RECEIVER)   ! start sending data
  CALL SHMEM_QUIET                              ! wait for delivery
  CALL SHMEM_PUT(FLAG_VAR, 1, 1, RECEIVER)      ! send completion flag
ELSE IF (MY_PE() .EQ. RECEIVER) THEN
  CALL SHMEM_UDCFLUSH
  CALL SHMEM_WAIT(FLAG_VAR, 0)
  PRINT*,ARRAY                  ! ARRAY has been delivered
ENDIF
END
```

SEE ALSO

intro_shmem(3), shmem_barrier(3), shmem_put(3), shmem_wait(3)

cpp(1)

barrier(3C), barrier(3F), BARSYNC(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

shmem_short_g, shmem_int_g, shmem_long_g, shmem_float_g, shmem_double_g – transfers one data item from a remote processor

SYNOPSIS

C:

```
#include <mpp/shmem.h>
short shmem_short_g(short *addr, int pe);
int shmem_int_g(int *addr, int pe);
long shmem_long_g(long *addr, int pe);
float shmem_float_g(float *addr, int pe);
double shmem_double_g(double *addr, int pe);
```

IMPLEMENTATION

C routines:

Cray MPP systems
Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

These routines provide a very low latency GET capability for single elements of most basic types.

These routines have the following arguments:

addr The remotely accessible array element or scalar data object.

pe The number of the remote processing element (PE) on which *addr* resides.

NOTES

These functions are provided in C in the form of macros on CRAY T3E and Cray PVP systems.

SEE ALSO

`intro_shmem`(3), `shmem_get`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_p, shmem_int_p, shmem_long_p, shmem_float_p, shmem_double_p – transfers one data item to a remote processor

SYNOPSIS

C:

```
#include <mpp/shmem.h>
void shmem_short_p(short *addr, short value, int pe);
void shmem_int_p(int *addr, int value, int pe);
void shmem_long_p(long *addr, long value, int pe);
void shmem_float_p(float *addr, float value, int pe);
void shmem_double_p(double *addr, double value, int pe);
```

IMPLEMENTATION

C routines:
Cray MPP systems
Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

These routines provide a very low latency PUT capability for single elements of most basic types.

These routines have the following arguments:

addr The remotely accessible array element or scalar data object which will receive the data on the remote processing element(PE).

value The value to be transfered to *addr* on the remote PE.

pe The number of the remote PE.

As with shmem_put(3), these functions start the remote transfer and may return before the data is delivered to the remote PE. Use shmem_quiet(3) or barrier(3F) to force completion of all remote PUT transfers.

NOTES

These functions are provided in C in the form of macros on CRAY T3E and Cray PVP systems.

SEE ALSO

`intro_shmem`(3), `shmem_put`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_stack – Makes a stack address remotely accessible

SYNOPSIS

C:

```
void shmem_stack(void *stack_var);
```

Fortran (CF77 and CF90):

```
CALL SHMEM_STACK(stack_var)
```

IMPLEMENTATION

CRAY T3D systems

DESCRIPTION

The shmem_stack routine makes a stack address on a remote processing element (PE) remotely accessible from the local PE. This is done by checking that the *stack_var* address is a valid stack address on the current PE. If *stack_var* is beyond the top of stack, shmem_stack extends the stack to include *stack_var*.

The shmem_stack routine can be called prior to shmem_put(3) or shmem_get(3) when the *stack_var* address is on the stack. This ensures that an operand range error does not occur during the data-passing step. For shmem_get(3) or shmem_put(3) to work properly, a remote *stack_var* address must also be a valid address on the local PE.

The shmem_stack routine accepts the following argument:

stack_var Data object of any noncharacter type. It must reside on the private stack.

NOTES

For more information about remotely accessible data objects, see intro_shmem(3).

SEE ALSO

intro_shmem(3), shmem_get(3), shmem_put(3)

NAME

shmem_short_sum_to_all, shmem_int_sum_to_all, shmem_float_sum_to_all, shmem_double_sum_to_all, shmem_complexf_sum_to_all, shmem_complexd_sum_to_all, shmem_int8_sum_to_all, shmem_real8_sum_to_all, shmem_comp8_sum_to_all, shmem_int4_sum_to_all, shmem_real4_sum_to_all, shmem_comp4_sum_to_all – Performs a sum reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_sum_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride, int PE_size,
   short *pWrk, long *pSync);

void shmem_int_sum_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);

void shmem_float_sum_to_all(float *target, float *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, float *pWrk, long *pSync);

void shmem_double_sum_to_all(double *target, double *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, double *pWrk, long *pSync);

void shmem_complexf_sum_to_all(float complex *target,
   float complex *source, int nreduce, int PE_start,
   int logPE_stride, int PE_size, float complex *pWrk,
   long *pSync);
```

```
void shmem_complexd_sum_to_all(double complex *target,
  double complex *source, int nreduce, int PE_start,
  int logPE_stride, int PE_size, double complex *pWrk,
  long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

INTEGER target(nreduce), source(nreduce)
INTEGER pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_INT8_SUM_TO_ALL(target, source, nreduce, PE_start,
     logPE_stride, PE_size, pWrk, pSync)

REAL target(nreduce), source(nreduce)
REAL pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_REAL8_SUM_TO_ALL(target, source, nreduce, PE_start,
     logPE_stride, PE_size, pWrk, pSync)

COMPLEX target(nreduce), source(nreduce)
COMPLEX pWrk(MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_COMP8_SUM_TO_ALL(target, source, nreduce, PE_start,
     logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

INTEGER(KIND=4) target(nreduce), source(nreduce)
INTEGER(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_INT4_SUM_TO_ALL(target, source, nreduce, PE_start,
     logPE_stride, PE_size, pWrk, pSync)

REAL(KIND=4) target(nreduce), source(nreduce)
REAL(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_REAL4_SUM_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

COMPLEX(KIND=4) target(nreduce), source(nreduce)
COMPLEX(KIND=4) pWrk(MAX(nreduce/2+1,
   SHMEM_REDUCE_MIN_WRKDATA_SIZE))
CALL SHMEM_COMP4_SUM_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple processing elements (PEs). A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_sum_to_all`	Complex. If you are using the CF90 compiler, it must be a default complex value.
`shmem_comp4_sum_to_all`	`COMPLEX(KIND=4).`
`shmem_int8_sum_to_all`	Integer. If you are using the CF90 compiler, it must be a default integer value.
`shmem_int4_sum_to_all`	`INTEGER(KIND=4).`
`shmem_real8_sum_to_all`	Real. If you are using the CF90 compiler, it must be a default real value.
`shmem_real4_sum_to_all`	`REAL(KIND=4).`

source
A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.
- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and finds the sum of the real variable `FOO` across all even PEs.

```
  INCLUDE "mpp/shmem.fh"

  INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
  DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
  PARAMETER (NR=1)
  REAL FOO, FOOSUM, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
  COMMON /COM/ FOO, FOOSUM, PWRK
  INTRINSIC MY_PE

  IF ( MOD(MY_PE(),2) .EQ. 0) THEN
     CALL SHMEM_INT4_SUM_TO_ALL(FOOSUM, FOO, NR, 0, 1, N$PES/2,
 &       PWRK, PSYNC)
     PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOSUM
  ENDIF
```

Example 2: Consider the following C call:

```
shmem_int_sum_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_int_sum_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_int_sum_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_int_sum_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

intro_shmem(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_swap, shmem_short_swap, shmem_int_swap, shmem_long_swap, shmem_float_swap, shmem_double_swap – Performs an atomic swap to a remote data object

SYNOPSIS

C or C++:

```
#include <mpp/shmem.h>

long shmem_swap(long *target, long value, int pe);
short shmem_short_swap(short *target, short value, int pe);
int shmem_int_swap(int *target, int value, int pe);
long shmem_long_swap(long *target, long value, int pe);
float shmem_float_swap(float *target, float value, int pe);
double shmem_double_swap(double *target, double value, int pe);
```

Fortran:

```
INTEGER pe

INTEGER SHMEM_SWAP
ires = SHMEM_SWAP(target, value, pe)

INTEGER*4 SHMEM_INT4_SWAP
ires = SHMEM_INT4_SWAP(target, value, pe)

INTEGER*8 SHMEM_INT8_SWAP
ires = SHMEM_INT8_SWAP(target, value, pe)

REAL*4 SHMEM_REAL4_SWAP
res = SHMEM_REAL4_SWAP(target, value, pe)

REAL*8 SHMEM_REAL8_SWAP
res = SHMEM_REAL8_SWAP(target, value, pe)
```

IMPLEMENTATION

CRAY T3D systems

CRAY T3E systems

DESCRIPTION

`shmem_swap` performs an atomic swap operation. It writes value *value* into *target* on processing element (PE) *pe* and returns the previous contents of *target* as an atomic operation.

The arguments are as follows:

target The remotely accessible integer data object to be updated on the remote PE. If you are using C, the type of *target* should match that implied in the SYNOPSIS section. If you are using the CF90 compiler, it must be of the following type.

`SHMEM_SWAP`	8-byte INTEGER
`SHMEM_INT4_SWAP`	4-byte INTEGER
`SHMEM_INT8_SWAP`	8-byte INTEGER
`SHMEM_REAL4_SWAP`	4-byte REAL
`SHMEM_REAL8_SWAP`	8-byte REAL

value Value to be atomically written to the remote PE. *value* is the same type as *target*.

pe An integer that indicates the PE number on which *target* is to be updated. If you are using the CF90 compiler, it must be a default integer value.

NOTES

The term remotely accessible is defined in `intro_shmem`(3).

`shmem_swap` writes directly into another PE's memory without notifying that PE, so the receiving PE should normally set automatic cache invalidation by use of `shmem_set_cache_inv`(3) or `shmem_set_cache_line_inv`(3). If this is not done, the receiving PE's data cache might become incoherent.

You can use shmem_swap without setting the remote write invalidate bit if you know, prior to the call, that the data at the target addresses are not cachable or are otherwise known not to reside in the remote processor's cache at the time of the call. Alternatively, the receiving processor can flush its cache when it knows that it has received data from another processor. In general, this requires synchronization. The cache can be flushed using shmem_udcflush(), shmem_udcflush_line(), the #pragma _CRI suppress directive (from C), or the CDIR$ SUPPRESS directive (from Fortran).

RETURN VALUES

The contents that had been at the target address on the remote PE prior to the swap is returned.

SEE ALSO

intro_shmem(3), shmem_clear_cache_inv(3), shmem_set_cache_inv(3), shmem_set_cache_line_inv(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

NAME

shmem_wait, shmem_wait_until – Waits for a variable on the local processing element (PE) to change

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_wait( long *ivar, long cmp_value )

void shmem_wait_until( long *ivar, int cmp, long value )
```

Fortran:

```
include ´mpp/shmem.fh´

INTEGER ivar, cmp, cmp_value

CALL SHMEM_WAIT(ivar, cmp_value)

CALL SHMEM_WAIT_UNTIL(ivar, cmp, cmp_value)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

shmem_wait and shmem_wait_until wait for *ivar* to be changed by a remote write or atomic swap issued by a different processor. These routines can be used for point-to-point directed synchronization. A call to shmem_wait does not return until some other processor writes a value, not equal to *cmp_value*, into *ivar* on the waiting processor. A call to shmem_wait_until does not return until some other processor changes *ivar* to satisfy the condition implied by *cmp* and *cmp_value*. This mechanism is useful when a processor needs to tell another processor that it has completed some action.

The arguments are as follows:

ivar A remotely accessible integer variable that is being updated by a remote processor. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *ivar* should match that implied in the SYNOPSIS section.

cmp The compare operator that compares *ivar* with *cmp_value*. *cmp* must be of type integer. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *cmp* should match that implied in the SYNOPSIS section.

The following *cmp* values are supported:

cmp **Value**	**Comparison**
`SHMEM_CMP_EQ` (Fortran), `_SHMEM_CMP_EQ` (C)	Equal
`SHMEM_CMP_NE` (Fortran), `_SHMEM_CMP_NE` (C)	Not equal
`SHMEM_CMP_GT` (Fortran), `_SHMEM_CMP_GT` (C)	Greater than
`SHMEM_CMP_LE` (Fortran), `_SHMEM_CMP_LE` (C)	Less than or equal to
`SHMEM_CMP_LT` (Fortran), `_SHMEM_CMP_LT` (C)	Less than
`SHMEM_CMP_GE` (Fortran), `_SHMEM_CMP_GE` (C)	Greater than or equal to

cmp_value
cmp_value must be of type integer. If you are using the CF90 compiler, it must be of default kind. If you are using C, the type of *cmp_value* should match that implied in the SYNOPSIS section.

The `shmem_wait` routine returns when *ivar* is no longer equal to *cmp_value*.

The `shmem_wait_until` routine returns when the compare condition is true. The compare condition is defined by the *ivar* argument compared with the *cmp_value* using the comparison operator, *cmp*.

NOTES

`shmem_wait` and `shmem_wait_until` perform the following functions:

- Invalidate the cache line associated with *ivar*
- Update the cache invalidation filter so that remote writes to this location will invalidate the corresponding cache line
- Wait, by looping in cache, until the value of *ivar* satisfies the wait condition

It may be more efficient to write a simple spin-wait loop with a variable that is declared or cast volatile. This is the case if other processors will not be writing blocks of data into the waiting processor. On CRAY T3D systems, it takes SHMEM_WAIT() about 100 clock periods (CPs) to set up the cache invalidation filter and another 100 CPs to disable the filter before returning to the caller.

EXAMPLES

Example 1: The following call returns when variable *ivar* is not equal to 100:

```
CALL SHMEM_WAIT(ivar, 100)
```

Example 2: The following call to SHMEM_WAIT_UNTIL is equivalent to the call to SHMEM_WAIT in example 1:

```
CALL SHMEM_WAIT_UNTIL(ivar, SHMEM_CMP_NE, 100)
```

Example 3: The following C call waits until the sign bit in ivar is set by a transfer from a remote PE:

```
shmem_wait_until(&ivar, _SHMEM_CMP_LT, 0);
```

Example 4: The following Fortran example is in the context of a subroutine:

```
SUBROUTINE EXAMPLE()
INTEGER FLAG_VAR
COMMON/FLAG/FLAG_VAR
. . .
FLAG_VAR = FLAG_VALUE    !  initialize the event variable
. . .
IF (FLAG_VAR .EQ.  FLAG_VALUE) THEN
        CALL SHMEM_WAIT(FLAG_VAR, FLAG_VALUE)
ENDIF
FLAG_VAR = FLAG_VALUE    !  reset the event variable for next time
. . .
END
```

SEE ALSO

intro_shmem(3), shmem_put(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

shmem_short_xor_to_all, shmem_int_xor_to_all, shmem_int8_xor_to_all, shmem_real8_xor_to_all, shmem_comp8_xor_to_all, shmem_int4_xor_to_all, shmem_real4_xor_to_all, shmem_comp4_xor_to_all – Performs a reduction across a set of processing elements (PEs)

SYNOPSIS

C:

```
#include <mpp/shmem.h>

void shmem_short_xor_to_all(short *target, short *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, short *pWrk, long *pSync);

void shmem_int_xor_to_all(int *target, int *source,
   int nreduce, int PE_start, int logPE_stride,
   int PE_size, int *pWrk, long *pSync);
```

Fortran (CF77 and CF90):

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT8_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL8_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_COMP8_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

Fortran 90 (CF90) only:

```
INCLUDE "mpp/shmem.fh"

INTEGER pSync(SHMEM_REDUCE_SYNC_SIZE)
INTEGER nreduce, PE_start, logPE_stride, PE_size

CALL SHMEM_INT4_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_REAL4_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)

CALL SHMEM_COMP4_XOR_TO_ALL(target, source, nreduce, PE_start,
   logPE_stride, PE_size, pWrk, pSync)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The shared memory (SHMEM) reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction is the result of performing an associative binary operation across a set of values. For a list of other SHMEM reduction routines, see intro_shmem(3).

The *nreduce* argument determines the number of separate reductions to perform. The source array on all PEs in the active set provides one element for each reduction. The results of the reductions are placed in the target array on all PEs in the active set. The active set is defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

The *source* and *target* arrays may be the same array, but they may not be overlapping arrays.

The arguments are as follows:

target A symmetric array of length *nreduce* elements to receive the results of the reduction operations.

The data type of *target* varies with the version of the reduction routine being called and the language used. When calling from C, refer to the SYNOPSIS section for data type information. When calling from Fortran, the *target* data types are as follows:

Routine	Data Type
`shmem_comp8_xor_to_all`	Complex, with an element size equal to two 8-byte real values
`shmem_comp4_xor_to_all`	Complex, with an element size equal to two 4-byte real values
`shmem_int8_xor_to_all`	Integer, with an element size of 8 bytes
`shmem_int4_xor_to_all`	Integer, with an element size of 4 bytes
`shmem_real8_xor_to_all`	Real, with an element size of 8 bytes
`shmem_real4_xor_to_all`	Real, with an element size of 4 bytes

source
A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction operation. The *source* argument must have the same data type as *target*.

nreduce
The number of elements in the *target* and *source* arrays. *nreduce* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_start
The lowest virtual PE number of the active set of PEs. *PE_start* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

logPE_stride
The log (base 2) of the stride between consecutive virtual PE numbers in the active set. *logPE_stride* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

PE_size
The number of PEs in the active set. *PE_size* must be of type integer. If you are using the CF90 compiler, it must be a default integer value.

pWrk
A symmetric work array. The *pWrk* argument must have the same data type as *target*.

In C, this contains
`max(`*nreduce*`/2 + 1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

In Fortran, this contains
`max(`*nreduce*`/2 + 1, SHMEM_REDUCE_MIN_WRKDATA_SIZE)` elements.

pSync
A symmetric work array.

In C, *pSync* is of type `int` and size `_SHMEM_REDUCE_SYNC_SIZE`.

In Fortran, *pSync* is of type integer and size `SHMEM_REDUCE_SYNC_SIZE`. If you are using the CF90 compiler, it must be a default integer value.

Every element of this array must be initialized with the value `_SHMEM_SYNC_VALUE` (in C) or `SHMEM_SYNC_VALUE` (in Fortran) before any of the PEs in the active set enter the reduction routine.

The values of arguments *nreduce*, *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the active set. The same *target* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the active set.

Upon return from a reduction routine, the following are true for the local PE:

- The *target* array is updated.
- The data cache region mapped to *target* is coherent.
- The values in the *pSync* array are restored to the original values.

NOTES

The terms *collective* and *symmetric* are defined in `intro_shmem`(3).

All SHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the active set while any of the PEs participate in processing of a SHMEM reduction routine. Be careful to avoid these situations:

- If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter a SHMEM routine called with the *pSync* synchronization array.

- A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the active set are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization. However, in the special case of reduction routines being called with the same active set, you can allocate two *pSync* and *pWrk* arrays and alternate between them on successive calls.

The reduction routines get highest performance if the number of PEs in the active set is a power of two and if the *pWrk* and *pSync* arrays are cache aligned.

EXAMPLES

Example 1: This CF77 example statically initializes the *pSync* array and computes the exclusive OR of variable FOO across all even PEs.

```
 INCLUDE "mpp/shmem.fh"

 INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
 DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
 PARAMETER (NR=1)
 REAL FOO, FOOXOR, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
 COMMON /COM/ FOO, FOOXOR, PWRK
 INTRINSIC MY_PE

 IF ( MOD(MY_PE(),2) .EQ. 0) THEN
    CALL SHMEM_REAL8_XOR_TO_ALL(FOOXOR, FOO, NR, 0, 1, N$PES/2,
&       PWRK, PSYNC)
    PRINT*,'Result on PE ´,MY_PE(),' is ´,FOOXOR
 ENDIF
```

Example 2: Consider the following C call:

```
shmem_short_xor_to_all( target, source, 3, 0, 0, 8, pwrk, psync );
```

The preceding call is more efficient, but semantically equivalent to, the combination of the following calls:

```
shmem_short_xor_to_all(&(target[0]), &(source[0]), 1, 0, 0, 8,
   pwrk1, psync1);
shmem_short_xor_to_all(&(target[1]), &(source[1]), 1, 0, 0, 8,
   pwrk2, psync2);
shmem_short_xor_to_all(&(target[2]), &(source[2]), 1, 0, 0, 8,
   pwrk1, psync1);
```

Note that two sets of *pWrk* and *pSync* arrays are used alternately because no synchronization is done between calls.

SEE ALSO

`intro_shmem`(3)

Cray MPP Fortran Reference Manual, publication SR–2504

Cray Standard C Reference Manual for MPP, publication SR–2506

Cray Research MPP Software Guide, publication SR–2508

NAME

start_pes – Starts a processing element (PE) for a message- or data-passing program

SYNOPSIS

C:

```
void start_pes(int npes)
```

Fortran:

```
CALL START_PES(npes)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The start_pes routine starts up additional tasks that enter at a main program. This routine starts multitasked tasks, also called PEs, for use in stand-alone shared memory (SHMEM), Parallel Virtual Machine (PVM), or Message Passing Interface (MPI) applications.

The start_pes routine can be called only once by the master task, or PE 0. It is a no-op if any other PE calls this function. This makes it convenient to simply place a call to start_pes at the first line of the main program to set up a program to be a message-passing or data-passing application.

The start_pes routine accepts the following argument:

npes Identifies the total number of PEs desired. If *npes* is 0, then the number of PEs is selected by the NPES environment variable.

EXAMPLES

This is a simple two-PE program that calls shmem_put(3):

```
PROGRAM PUT

INTEGER TARG, SRC, RECEIVER, BAR
TASK COMMON /T/ TARG
PARAMETER (RECEIVER=1)
PARAMETER (NPES=2)
INTEGER SHMEM_MY_PE

CALL START_PES(NPES)

IF (SHMEM_MY_PE() .EQ. 0) THEN
        SRC = 33
        CALL SHMEM_PUT(TARG, SRC, 1, RECEIVER)
ENDIF

CALL BARRIER       ! synchronize sender and receiver

IF (SHMEM_MY_PE() .EQ. RECEIVER) THEN
        CALL SHMEM_UDCFLUSH()
        PRINT*,'PE ',SHMEM_MY_PE(),' TARG=',TARG
        IF (TARG.NE.33) PRINT*,'FAIL'
ENDIF
END
```

NOTES

The start_pes routine uses a barrier to synchronize the newly started tasks before any of the new tasks enter the main program.

SEE ALSO

intro_shmem(3), num_pes(3), my_pe(3I), shmem_put(3)

Message Passing Toolkit: PVM Programmer's Manual, publication SR–2196

NAME

INTRO_SYNC – Introduction to synchronization routines

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

Synchronization routines are used to synchronize parallel tasks within programs. This section describes a set of synchronization primitives and timing routines that are available on Cray MPP systems. For further information on using these routines, see the *Cray MPP Fortran Reference Manual*, publication SR–2504. For information about multitasking on systems other than Cray MPP systems, see the *UNICOS System Libraries Reference Manual*, publication SR–2080.

All synchronization routines run on Cray MPP systems only.

MPP Definitions

This section provides a brief description of terms used to describe synchronization routines.

Variables can either be *private* or *shared*. Private variables can only be accessed by the task that created them. Shared variables are accessible by all tasks. By default, all variables are considered private. A variable becomes shared only if it has been specified on a SHARED compiler directive. For more information, see the *Cray MPP Fortran Reference Manual*, publication SR–2504.

Cray MPP systems implement a special hardware barrier network to implement two types of barrier synchronization operations: *barriers* and *eurekas*. A barrier is a point within a program where a task must wait until all other tasks reach the barrier. A eureka is a point within a program where all tasks are informed when one task has satisfied some condition. Eureka synchronization has several uses, including database searches. Using eureka synchronization, a programmer can stop a database search as soon as any task finds the data rather than waiting for all of the tasks to exhaust the search. For more information about the hardware barrier network, see the *CRAY T3D System Architecture Overview*, publication HR–04033.

MPP Lock Routines

Lock routines are used to serialize access to shared variables. The following are the lock routines:

- SET_LOCK: sets a lock
- CLEAR_LOCK: clears a lock
- TEST_LOCK: sets a lock and returns the prior state of a lock

Unlike systems other than Cray MPP systems, there is no need to assign a lock via a routine such as LOCKASGN. For Cray MPP systems, the user only needs to insure that the shared lock variable be initialized to zero before using it as a lock control variable.

MPP Event Routines

Event routines are used to record the state of a program's execution and to communicate that state to another task. The event routines can be used in one of two modes: memory mode or eureka mode. If a shared event variable is passed to an event routine, that shared variable is used for event communication and the event routine is said to be used used in memory mode. If an argument is not supplied to an event routine, then the eureka mechanism in the Cray MPP hardware barrier network is used for event communication and the event routine is said to be used in eureka mode.

The following are the event routines:

- SET_EVENT: sets an event
- WAIT_EVENT: suspends task execution until an event is set
- CLEAR_EVENT: clears a memory mode event or arms the eureka mode event
- TEST_EVENT: returns the state of an event

Unlike systems other than Cray MPP systems, there is no need to assign an event variable via a routine such as EVASGN. For Cray MPP systems, the user only needs to ensure that the shared event variable be initialized to zero before using it as an event control variable.

MPP Barrier Routines

Barrier routines are used to synchronize the execution of all tasks. A task's execution is suspended at a barrier until all tasks have reached the barrier. The barrier routines use the Cray MPP hardware barrier network. For a description of this network, see the *CRAY T3D System Architecture Overview*, publication HR–04033.

The following are the barrier routines:

- `BARRIER`: registers the arrival of a task at the barrier then waits for all other tasks to arrive
- `SET_BARRIER`: registers the arrival of a task at the barrier
- `WAIT_BARRIER`: waits for all other tasks to arrive at the barrier
- `TEST_BARRIER`: returns the state of the barrier

When adding a barrier to your program, be sure that all tasks will eventually reach the barrier. If any of the tasks do not reach the barrier because of task-dependent conditions or branches, the other tasks will wait at the barrier indefinitely or abort with a user deadlock error.

SEE ALSO

CRAY T3D System Architecture Overview, publication HR–04033

Cray MPP Fortran Reference Manual, publication SR–2504

UNICOS System Libraries Reference Manual, publication SR–2080

NAME

barrier – Registers the arrival of a task at a barrier and suspends task execution until all other tasks arrive at the barrier

SYNOPSIS

```
void barrier()
```

IMPLEMENTATION

Cray MPP systems

Message Passing Toolkit (MPT) on Cray PVP systems

DESCRIPTION

The barrier function registers the arrival of a task at a barrier. Barriers are a fast mechanism for synchronizing all tasks at once. barrier causes a task to suspend execution until all tasks have called either barrier or set_barrier(3C).

On Cray PVP systems, the barrier function must be used with tasks started by start_pes(3), or pvm_spawn(3) using the MtSpawn or MtSpawnSelf flags. The barrier function is not supported with Autotasking or macrotasking.

EXAMPLES

```
setup_data()
{
      if (_my_pe() == 0) {
          setup();
      }

      /*
       * Force all tasks to wait for PE 0 to complete
       * routine setup().
       */

      barrier();
}
```

SEE ALSO

`set_barrier`(3C), `start_pes`(3), `test_barrier`(3C), `wait_barrier`(3C)

NAME

clear_event – Clears the eureka event and returns control to the calling task

SYNOPSIS

```
void  clear_event(void)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The clear_event function clears the eureka event and returns control to the calling task. The result of this is that tasks subsequently performing wait_event(3C) calls must wait. If a posted eureka event is not cleared, the posted condition remains outstanding.

When the event routines are used, clear_event initializes the eureka barrier bit to the cleared state. All tasks must call clear_event before any task can test, wait, or post the eureka mode event. If any tasks fail to call clear_event, the program's behavior is undefined. Although all tasks must call clear_event when using the event routines, tasks are not required to wait for the eureka event to be posted. However, before another eureka activity can be started, all tasks must once again call clear_event to reinitialize the eureka barrier bit.

EXAMPLES

The following code shows the use of the `clear_event` function.

```
search_data()
{
        long flag;

        /*
         * All PEs must call clear_event() to initialize the eureka barrier.
         */

        clear_event();

        /*
         * The first processor to return from search with a return
         * value of 0, will post the eureka event.
         */

        while ((flag = search()) != -1) {

            /*
             * If the event is posted, exit the while loop.
             */

            if (test_event()) {
                flag = -1;
                break;
            }

            /*
             * If the search is successful, post the event.
             */

            if (flag == 0) {
                set_event();
                break;
            }
        }
}
```

SEE ALSO

`set_event`(3C), `test_event`(3C), `wait_event`(3C)

NAME

CLEAR_EVENT – Clears an event and returns control to the calling task

SYNOPSIS

CALL CLEAR_EVENT([*event*])

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The argument to CLEAR_EVENT is optional. If an argument is supplied, it must be a SHARED integer variable. This variable will be used for event communication. When calling the event routines with a SHARED argument, the event routines are operating in *memory mode*. Memory mode events are relatively slow when compared with eureka mode (described below) but memory mode events are very flexible to use; for example, you can have several memory mode events active at any one time, but there can be only one eureka mode event active at a time.

When the posting of one memory mode event is required (a simple signal), call CLEAR_EVENT immediately after WAIT_EVENT to indicate that the posting of the event was detected.

When no event variable is specified, the routine operates in *eureka mode*. In eureka mode, the hardware barrier network is used for event communication, and the order in which you call WAIT_EVENT and CLEAR_EVENT is reversed from their use in memory mode (that is, all tasks must clear a eureka event before any tasks can test, wait for, or post the event). All tasks do not have to wait for a eureka event to be posted. However, all tasks must once again clear the event before another eureka activity can begin.

event Optional. Name of a SHARED integer variable used for event communication. This variable must be initialized to 0 before it is used.

EXAMPLES

Memory mode example:

```
      PROGRAM MULTI
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
CDIR$ MASTER
      EVSTART = 0
CDIR$ ENDMASTER
      IF (MY_PE() .NE. 0) THEN
          CALL MULTI2()
      ELSE
C      ...
          CALL SET_EVENT(EVSTART)
      ENDIF
      END

      SUBROUTINE MULTI2
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
C     ...
      CALL WAIT_EVENT(EVSTART)
      IF (HOME(EVSTART) .NE. MY_PE()) THEN
          CALL CLEAR_EVENT(EVSTART)
      ENDIF
C     ...
      END
```

Eureka mode example:

```
          PROGRAM MULTI
C         ...
          CALL CLEAR_EVENT()
          DO I=1, MAXVAL
              CALL GINK(RESULT,A(I,1))
              IF (RESULT .EQ. SEARCH_VAL) THEN
               CALL SET_EVENT();
               GOTO 20
            ENDIF
          IF (TEST_EVENT()) GOTO 20
          ENDDO
          CALL WAIT_EVENT()
20        CONTINUE
C         ...
          END
```

SEE ALSO

INTRO_SYNC(3F), SET_EVENT(3F), WAIT_EVENT(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

CLEAR_LOCK – Clears a lock and returns control to the calling task

SYNOPSIS

CALL CLEAR_LOCK(*lock*)

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

CLEAR_LOCK clears a lock and returns control to the calling task. Clearing the lock can allow another task to resume execution, but this is transparent to the task calling CLEAR_LOCK.

lock Name of a SHARED integer variable used as a lock. This variable must be initialized to 0 before it is used.

EXAMPLES

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      COMMON/LKCOM/LKOUTPUT
CDIR$ SHARED LKOUTPUT
CDIR$ MASTER
      LKOUTPUT = 0
CDIR$ ENDMASTER
C     ...
      CALL SET_LOCK(LKOUTPUT)
C     ...
C     <CRITICAL SECTION OF CODE>
C     ...
      CALL CLEAR_LOCK(LKOUTPUT)
C
C     ...
      END
```

SEE ALSO

INTRO_SYNC(3F), SET_LOCK(3F), TEST_LOCK(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

`set_barrier` – Registers the arrival of a task at a barrier

SYNOPSIS

```
void  set_barrier()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

Tasks can spend time waiting at a barrier. To spend task time at a barrier productively, you can create your own barrier mechanism using `set_barrier` and `wait_barrier`. `set_barrier` indicates that the calling task has arrived at the barrier; `wait_barrier` suspends execution of the calling task until all of the other tasks have arrived at the barrier.

You can place code between the two calls that will execute while waiting for other tasks to arrive at the barrier. In this way, a task can continue to do useful work after notifying that it has reached the barrier. When it completes the extra work, the task calls `wait_barrier` and waits, if necessary, for the remaining tasks to reach the barrier.

SEE ALSO

`barrier`(3C), `test_barrier`(3C), `wait_barrier`(3C)

NAME

SET_BARRIER – Registers the arrival of a task at a barrier

SYNOPSIS

```
CALL  SET_BARRIER()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

Tasks can spend time waiting at a barrier. To spend task time at a barrier productively, you can create your own barrier mechanism using SET_BARRIER and WAIT_BARRIER. SET_BARRIER indicates that the calling task has arrived at the barrier; WAIT_BARRIER suspends execution of the calling task until all of the other tasks have arrived at the barrier.

You can place code between the two calls that will execute while waiting for other tasks to arrive at the barrier. In this way, a task can continue to do useful work after notifying that it has reached the barrier. When it completes the extra work, the task calls WAIT_BARRIER and waits, if necessary, for the remaining tasks to reach the barrier.

SEE ALSO

BARRIER(3F), INTRO_SYNC(3F), TEST_BARRIER(3F), WAIT_BARRIER(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

SET_EVENT – Posts an event and returns control to the calling task

SYNOPSIS

```
CALL SET_EVENT([event])
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The argument to SET_EVENT is optional. If an argument is supplied, it must be a SHARED integer variable. This variable will be used for event communication. When calling the event routines with a SHARED argument, the event routines are operating in *memory mode*. Memory mode events are relatively slow when compared with eureka mode (described below) but memory mode events are very flexible to use; for example, you can have several memory mode events active at any one time, but there can be only one eureka mode event active at a time.

When the setting of one memory mode event is required (a simple signal), call CLEAR_EVENT immediately after WAIT_EVENT to indicate that the setting of the event was detected.

When no event variable is specified, the routine operates in *eureka mode*. In eureka mode, the hardware barrier network is used for event communication, and the order in which you call WAIT_EVENT and CLEAR_EVENT is reversed from their use in memory mode (that is, all tasks must clear a eureka event before any tasks can test, wait for, or post the event). All tasks do not have to wait for a eureka event to be posted. However, all tasks must once again clear the event before another eureka activity can begin.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

event Optional. Name of a SHARED integer variable used for event communication. This variable must be initialized to 0 before it is used.

EXAMPLES

Memory mode example:

```
      PROGRAM MULTI
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
CDIR$ MASTER
      EVSTART = 0
CDIR$ ENDMASTER
      If (MY_PE() .NE. 0) THEN
          CALL MULTI2()
      ELSE
C     ...
      CALL SET_EVENT(EVSTART)
      ENDIF
      END

      SUBROUTINE MULTI2
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
C     ...
      CALL WAIT_EVENT(EVSTART)
      IF (HOME(EVSTART) .NE. MY_PE()) THEN
          CALL CLEAR_EVENT(EVSTART)
      ENDIF
C     ...
      END
```

Eureka mode example:

```
      PROGRAM MULTI
C     ...
      CALL CLEAR_EVENT()
      DO I=1, MAXVAL
         CALL GINK(RESULT,A(I,1))
           IF (RESULT .EQ. SEARCH_VAL) THEN
               CALL SET_EVENT();
               GOTO 20
           ENDIF
           IF (TEST_EVENT()) GOTO 20
      ENDDO
      CALL WAIT_EVENT()
20    CONTINUE
C     ...
      END
```

SEE ALSO

CLEAR_EVENT(3F), INTRO_SYNC(3F), WAIT_EVENT(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

SET_LOCK – Sets a lock and returns control to the calling task

SYNOPSIS

CALL SET_LOCK(*lock*)

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

SET_LOCK sets a lock and returns control to the calling task. If the lock is already set when SET_LOCK is called, SET_LOCK waits until the lock is cleared by another task and then sets it. This means that placing SET_LOCK before a critical region ensures that the code in the region is executed only when the task has unique access to the lock. Calls to SET_LOCK cannot be nested.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

lock Name of a SHARED integer variable used as a lock. This variable must be initialized to 0 before it is used.

SEE ALSO

CLEAR_LOCK(3F), INTRO_SYNC(3F), TEST_LOCK(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

`test_barrier` – Tests a barrier to determine its state (set or cleared)

SYNOPSIS

```
long  test_barrier()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

This barrier routine checks to see if all tasks have arrived at the barrier.

return A logical .TRUE. if all processors have arrived and set the barrier. A logical .FALSE. if all processors have not set the barrier.

SEE ALSO

`barrier`(3C), `set_barrier`(3C), `wait_barrier`(3C)

NAME

TEST_BARRIER – Tests a barrier to determine its state (set or cleared)

SYNOPSIS

```
LOGICAL  TEST_BARRIER
return=TEST_BARRIER()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

return A logical .TRUE. if all tasks have arrived and set the barrier. A logical .FALSE. if all tasks have not arrived at the barrier.

NOTES

TEST_BARRIER and *return* must be declared type LOGICAL in the calling module.

SEE ALSO

BARRIER(3F), INTRO_SYNC(3F), SET_BARRIER(3F), WAIT_BARRIER(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

`test_event` – Returns the state of the eureka event, either posted or cleared

SYNOPSIS

```
long  test_event(void)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

return A logical true if the eureka event is posted. A logical false if the event is cleared. The eureka barrier bit's state is unaffected by a call to `test_event`.

Eureka events are potentially useful to signal the end of a search or other parallel activity where a single success should end the attempt by all tasks. For more information on the hardware eureka mechanism, see the *CRAY T3D System Architecture Overview*, publication HR–04033.

SEE ALSO

`clear_event`(3C), `set_event`(3C), `wait_event`(3C)

CRAY T3D System Architecture Overview, publication HR–04033

NAME

TEST_EVENT – Returns the state of an event

SYNOPSIS

```
LOGICAL  TEST_EVENT
return=TEST_EVENT([event])
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The argument to TEST_EVENT is optional. If an argument is supplied, it must be a SHARED integer variable. This variable will be used for event communication. When calling the event routines with a SHARED argument, the event routines are operating in *memory mode*. Memory mode events are relatively slow when compared with eureka mode (described following) but memory mode events are very flexible to use; for example, you can have several memory mode events active at any one time, but there can be only one eureka mode event active at a time.

When no event variable is specified, the routine operates in *eureka mode*. In eureka mode, the hardware barrier network is used for event communication. This implies that TEST_EVENT is returning the state of the eureka barrier bit.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

return A logical .TRUE. if the event is set, .FALSE. otherwise. The TEST_EVENT routine will return .FALSE. if the SHARED event control variable is in the busy state. An event control variable can be in the busy state if another PE is executing a call to SET_EVENT, WAIT_EVENT, or CLEAR_EVENT at the same time the call to TEST_EVENT occurs.

event Optional. Name of a SHARED integer variable used for event communication. This variable must be initialized to 0 before it is used.

NOTES

TEST_EVENT and *return* must be declared as type LOGICAL in the calling module.

SEE ALSO

CLEAR_EVENT(3F), INTRO_SYNC(3F), SET_EVENT(3F), WAIT_EVENT(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

TEST_LOCK – Sets a lock and returns the lock state

SYNOPSIS

```
LOGICAL  TEST_LOCK
return=TEST_LOCK(lock)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

This routine automatically sets a lock and returns the state that the lock was in (.TRUE. if set or .FALSE. if cleared) prior to the test. By using this function, a task can avoid blocking on a set lock by testing the lock.

return A logical .FALSE. if the lock was originally cleared. A logical .TRUE. otherwise. The TEST_LOCK routine will return .TRUE. if the SHARED lock control variable is in the busy state. A lock control variable can be in the busy state if another PE is executing a call to SET_LOCK, or CLEAR_LOCK at the same time the call to TEST_LOCK occurs.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

lock Name of a SHARED integer variable used as a lock. This variable must be initialized to 0 before it is used.

Unlike SET_LOCK(3F), the task calling TEST_LOCK does not wait for the lock to be unlocked. A task using TEST_LOCK must always test the return value before continuing.

NOTES

TEST_LOCK and *return* must be declared type LOGICAL in the calling module.

SEE ALSO

CLEAR_LOCK(3F), INTRO_SYNC(3F), SET_LOCK(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

`wait_barrier` – Suspends task execution until all tasks arrive at the barrier

SYNOPSIS

```
void  wait_barrier()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

`wait_barrier` suspends task execution until all tasks arrive at the barrier. Tasks signal their arrival at the barrier by calling either `barrier`() or `set_barrier`().

SEE ALSO

`barrier`(3C), `set_barrier`(3C), `test_barrier`(3C)

NAME

WAIT_BARRIER – Suspends task execution until all tasks arrive at the barrier

SYNOPSIS

```
CALL  WAIT_BARRIER()
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

Tasks can spend time waiting at a barrier. To spend task time at a barrier productively, you can create your own barrier mechanism using SET_BARRIER and WAIT_BARRIER. SET_BARRIER indicates that the calling task has arrived at the barrier; WAIT_BARRIER suspends execution of the calling task until all of the other tasks have arrived at the barrier.

You can place code between the two calls that will execute while waiting for other tasks to arrive at the barrier. In this way, a task can continue to do useful work after notifying that it has reached the barrier. When it completes the extra work, the task calls WAIT_BARRIER and waits, if necessary, for the remaining tasks to reach the barrier. WAIT_BARRIER suspends task execution until all tasks arrive at the barrier.

If WAIT_BARRIER is called from outside of a parallel region, the user program will deadlock. WAIT_BARRIER must be called from within a parallel region and all tasks must participate. The user can guard against calling WAIT_BARRIER from outside of a parallel region with the use of the IN_PARALLEL compiler intrinsic.

EXAMPLES

```
IF (IN_PARALLEL()) THEN
    CALL WAIT_BARRIER()
END IF
```

SEE ALSO

BARRIER(3F), INTRO_SYNC(3F), SET_BARRIER(3F), TEST_BARRIER(3F),

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

`wait_event` – Delays the calling task until the eureka event is posted

SYNOPSIS

```
void  wait_event(void)
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

`wait_event` suspends task execution at a cleared event until that event is posted by `set_event`. `wait_event` does not change the state of the event. `wait_event` always uses the hardware eureka mechanism for event communication.

The event routines use the MPP hardware eureka mechanism for event communication. For more information on the hardware eureka mechanism, see *CRAY T3D System Architecture Overview*, publication HR–04033.

SEE ALSO

`clear_event`(3C), `set_event`(3C), `test_event`(3C)

NAME

WAIT_EVENT – Delays the calling task until the specified event is posted

SYNOPSIS

CALL WAIT_EVENT([*event*])

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

WAIT_EVENT suspends task execution at a cleared event until that event is posted by SET_EVENT. WAIT_EVENT does not change the state of the event. The argument to WAIT_EVENT is optional. If an argument is supplied, it must be a SHARED integer variable. This variable will be used for event communication. When calling the event routines with a SHARED argument, the event routines are operating in *memory mode*. Memory mode events are relatively slow when compared with eureka mode (described below) but memory mode events are very flexible to use; for example, you can have several memory mode events active at any one time, but there can be only one eureka mode event active at a time.

When the posting of one memory mode event is required (a simple signal), call CLEAR_EVENT immediately after WAIT_EVENT to indicate that the posting of the event was detected.

When no event variable is specified, the routine operates in *eureka mode*. In eureka mode, the hardware barrier network is used for event communication, and the order in which you call WAIT_EVENT and CLEAR_EVENT is reversed from their use in memory mode (that is, all tasks must clear a eureka event before any tasks can test, wait for, or post the event). All tasks do not have to wait for a eureka event to be posted. However, all tasks must once again clear the event before another eureka activity can begin.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

event Optional. Name of a SHARED integer variable used for event communication. This variable must be initialized to 0 before it is used.

EXAMPLES

Memory mode example:

```
      PROGRAM MULTI
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
CDIR$ MASTER
      EVSTART = 0
CDIR$ ENDMASTER
      If (MY_PE() .NE. 0) THEN
          CALL MULTI2()
      ELSE
C     ...
      CALL SET_EVENT(EVSTART)
      ENDIF
      END

      SUBROUTINE MULTI2
      INTEGER EVSTART
      COMMON /EVCOM/ EVSTART
CDIR$ SHARED EVSTART
C     ...
      CALL WAIT_EVENT(EVSTART)
      IF (HOME(EVSTART) .NE. MY_PE()) THEN
          CALL CLEAR_EVENT(EVSTART)
      ENDIF
C     ...
      END
```

Eureka mode example:

```
      PROGRAM MULTI
C     ...
      CALL CLEAR_EVENT()
      DO I=1, MAXVAL
           CALL GINK(RESULT,A(I,1))
           IF (RESULT .EQ. SEARCH_VAL) THEN
               CALL SET_EVENT();
               GOTO 20
           ENDIF
           IF (TEST_EVENT()) GOTO 20
      ENDDO
      CALL WAIT_EVENT()
20    CONTINUE
C     ...
      END
```

SEE ALSO

CLEAR_EVENT(3F), INTRO_SYNC(3F), SET_EVENT(3F)

Cray MPP Fortran Reference Manual, publication SR–2504

NAME

INTRO_SPECIALIBS – Introduction to specialized libraries and miscellaneous routines

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This section contains descriptions of various specialized libraries and miscellaneous routines that are not included elsewhere in this manual.

Routines and Supported Machine Types

In the printed version of these routines, the man pages are grouped according to architecture type. The following list briefly describes this grouping:

- Common routines which run on all Cray Research systems:

ICEIL	IPXFARGC
MOV	PACK
PXFACCESS	PXFCHDIR
PXFCHMOD	PXFCHOWN
PXFCHROOT	PXFCONST
PXFCREAT	PXFFCNTL
PXFFILENO	PXFGETARG
PXFGETENV	PXFINTGET
PXFINTSET	PXFLINK
PXFOPEN	PXFRENAME
PXFRMDIR	PXFSTAT
PXFSTRGET	PXFSTRSET
PXFSTRUCTCREATE	PXFSTRUCTFREE
PXFUNAME	PXFUNLINK
SECONDR	SYSCLOCK
TIMEF	TRIMLEN
TSECND	UNPACK

- Routines that run only on Cray PVP systems:

```
AUXSTAT          BITVEC
BITVECM          BYT
DTTS             ICPUSED
MTTS             MVC
P32              P6460
SECOND           TREMAIN
TSDT             TSMT
UNITTS
```

- Routines that run only on Cray MPP systems: none

The following routines run on SPARC systems: SECOND, SECONDR, SYSCLOCK, TIMEF, PXFGETENV, PXFGETARG, IPXFARGC, PXFACCESS

CHARACTER FUNCTIONS

The following routines are used for character functions:

- CHAR: converts integer to character.
- ICHAR: converts character to integer.
- INDEX: determines index location of a character substring within a string.
- LEN: determines the length of a character string.
- LGE, LGT, LLE, LLT: compares strings lexically.

PACKING ROUTINES

The packing routines provide alternative ways to pack and unpack data into or out of Cray words. The following routines are used for packing:

- P32: packs 32-bit words into Cray 64-bit words.
- U32: unpacks 32-bit words from Cray 64-bit words.
- P6460: packs 60-bit words into Cray 64-bit words.
- U6064: unpacks 60-bit words from Cray 64-bit words.
- PACK: compresses stored data.
- UNPACK: expands stored data.

BYTE AND BIT MANIPULATION ROUTINES

Byte and bit manipulation routines move bytes and bits between variables and arrays, compare bytes, perform searches with a byte count as a search argument, and perform conversion on bytes.

The following routines are used for bit manipulation:

- `PUTBYT`: replaces a byte in a variable or an array with a specified value
- `IGTBYT`: extracts a byte from a variable
- `MOV`: moves bytes between variables or arrays
- `MOVBIT`: moves bits between variables or arrays
- `MVC`: moves characters between memory areas

TIMING ROUTINES

Timing routines include time and date routines, which produce the time and/or date in specified forms. The following are date and time routines:

- `CLOCK`: returns the current system clock time
- `DATE`: returns the current date
- `JDATE`: returns the current Julian date
- `ICPUSED`: gets CPU time in real-time clock values
- `RTC`, `IRTC`: returns real-time clock values
- `SECOND`: returns the elapsed CPU time (in seconds of real numbers) since the start of a program
- `SECONDR`: returns real-time clock values
- `TIMEF`: returns the elapsed wall-clock time since the initial call to `TIMEF`
- `TREMAIN`: returns the CPU time (in seconds of real numbers) remaining for the program

MISCELLANEOUS ROUTINES

The following routines are available only on Cray PVP systems, with the exception of `TRIMLEN`(3F), which is available on all Cray Research systems.

- `DTTS`: converts ASCII date and time to time stamp
- `MTTS`: converts machine time to time stamp

- TRIMLEN: returns the length of a character argument without counting trailing blanks
- TSDT: converts time stamps to ASCII date and time strings
- TSMT: converts time stamps to real-time clock values
- UNITTS: returns time-stamp units in specified standard time units

OTHER ROUTINES

AUXSTAT specifies the number of reads and writes to secondary data segments (SDS) that have occurred while accessing auxiliary array variables.

ICEIL returns the integer ceiling of a rational number.

LOC returns the memory address of a variable or an array.

POSIX ROUTINES

The POSIX FORTRAN 77 Language Interfaces Standard IEEE Std 1003.9-1992 (POSIX.a) defines a standardized interface for accessing the system services of IEEE Std 1003.1-1990 (POSIX.1), and support routines to access constructs not directly accessible with FORTRAN 77.

Only a subset of the routines described in this standard are currently available on Cray Research systems. For some routines, only a portion of the functionality described by the standard is currently implemented.

Many of the service interfaces defined in POSIX.1 require the use of aggregate data types (for example, structures) that do not map to FORTRAN 77 entities. POSIX.9 solves this problem by the use of data abstraction; by using additional subroutines to access and manipulate the aggregate data, the underlying data structures are hidden from the user. It is the responsibility of the Fortran programmer to maintain variables corresponding to the individual components of the actual implementation of the aggregate data, but the programmer does not need to know the details of the actual implementation of the aggregate. The basic model of this data abstraction is as follows:

- The programmer calls a subroutine PXFSTRUCTCREATE to "create" an instance of the desired aggregate data type; this subroutine returns a handle that the programmer subsequently uses in order to reference and/or manipulate the data.
- The programmer uses additional subroutines to load values into, or extract values out of, the aggregate data. These subroutines are passed the handle of the desired aggregate and the name of the specific component that is to be accessed. The programmer has direct control over only one component at a time.

- If an application passes information to the system, the PXF*type*SET subroutine is called once for each member before calling the system procedure. Currently, only PXFINTSET and PXFSTRSET are implemented.
- If an application needs to get information from the system, the PXF*type*GET subroutine is called once for each member after calling the system procedure. Currently, only PXFINTGET and PXFINTSET are implemented.
- When an instance of an aggregate is no longer required, a subroutine (PXFSTRUCTFREE) can be called to release it.
- When calling the actual system procedure, the calling sequence is equivalent to the C binding as shown in POSIX.1, except that a handle is used in place of the POSIX.1 structure (pointer) argument.

NAME

ICEIL – Returns integer ceiling of a rational number

SYNOPSIS

i=ICEIL(*j*, *k*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ICEIL is an integer function that returns the integer ceiling of a rational number formed by two integer parameters.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

j The numerator of a rational number.

k The denominator of a rational number.

The value of function *i* is the smallest integer larger than or equal to *j*/*k*.

NAME

IPXFARGC – Returns number of command-line arguments excluding the command name

SYNOPSIS

```
INTEGER  FUNCTION  IPXFARGC()
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

The function IPXFARGC returns the number of command-line arguments in the command used to invoke the executing program. The command name is not included in the number returned. A return value of 0 indicates that there are no command-line arguments other than the command name itself.

NAME

STRMOV, MOVBIT, MOVBITZ – Moves bytes or bits from one variable or array to another

SYNOPSIS

```
CALL STRMOV(src, isb, num, dest, idb)
CALL MOVBIT(src, isb, num, dest, idb)
CALL MOVBITZ(src, isb, num, dest, idb)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

STRMOV moves bytes from one variable or array to another. MOVBIT moves bits from one variable or array to another.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src Variable or array of any type except character, and of any length, containing the bytes or string of bits to be moved. Bytes are numbered from 1, beginning at the leftmost byte position of *src*. When using MOVBITZ the numbering begins at 0.

isb Starting byte or bit in the *src* string. Specify an integer variable, expression, or constant greater than 0. Bytes and bits are numbered from 1, beginning at the leftmost byte or bit position of *src*. *isb* is one-based for STRMOV and MOVBIT and zero-based for MOVBITZ.

num An integer variable, expression, or constant that contains the number of bytes or bits to be moved; it must be greater than 0.

dest Variable or array of any type except character, and of any length, that contains the starting byte or bit to receive the data. Bytes and bits are numbered from 1, beginning at the leftmost byte or bit position of *dest.*

idb An integer variable, expression, or constant that contains the starting byte or bit to receive the data; must be greater than 0. Bytes and bits are numbered from 1, beginning at the leftmost byte or bit position of *dest*. *idb* is one-based for STRMOV and MOVBIT and zero-based for MOVBITZ.

EXAMPLES

Cray MPP systems and Cray PVP systems:

```
      PROGRAM TEST
      INTEGER IBUFA(3),IA(3)
      DATA
     + IA/4918568664756134948,
     +    4938346680284645400,
     +    4688555661647431816/
*  MOVBIT 1: Move 129 bits from offset 8 to offset 5
      IBUFA=0
      CALL MOVBIT(IA(1),8,129,IBUFA,5)


*  MOVBIT 2: Move 100 bits from offset 35 to offset 10
      DO 202 J=1,3
202      IBUFA(J)=0
      CALL MOVBIT(IA(1),35,100,IBUFA,10)
*  STRMOV 1: Move 8 bytes from byte 5 to byte 1
      IBUFA=0
      CALL STRMOV(IA(1),5,8,IBUFA,1)
*  STRMOV 2: Move 8 bytes from offset 2 to offset 10
      IBUFA=0
      CALL STRMOV(IA(1),2,8,IBUFA,10)
   END
```

The results:

```
MOVBIT 1:
     IA= 042102210220221202204 4
         042210420220500610403 0
         040421061042021104421 0
     IBUFA= 00102210220221202204 42
            02210420220500610403 02
            00400000000000000000 00

MOVBIT 2:
     IA= 042102210220221202204 4
         042210420220500610403 0
         040421061042021104421 0
     IBUFA= 00004424044110422104 20
            04412014210060400000 00
            00000000000000000000 00

STRMOV 1:
     IA= 042102210220221202204 4
         042210420220500610403 0
         040421061042021104421 0
     IBUFA= 01105011022104421040 44
            00000000000000000000 00
            00000000000000000000 00

STRMOV 2:
     IA= 042102210220221202204 4
         042210420220500610403 0
         040421061042021104421 0
     IBUFA= 00000000000000000000 00
            00010221022022120220 44
            04200000000000000000 00
```

NAME

PACK – Compresses stored data

SYNOPSIS

CALL PACK(*p, nbits, u, count*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

PACK takes partial word items and packs them into 64-bit words.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

p On exit, vector of packed data.

nbits Number of rightmost bits of data in each partial word; must be 1, 2, 4, 8, 16, or 32.

u Vector of partial words to be compressed.

count Number of partial word items to be compressed.

PACK takes the 1, 2, 4, 8, 16, or 32 rightmost bits of several partial words and concatenates them into 64-bit words. The following equation gives the number of full words:

$$n = \frac{(count \cdot nbits)}{64}$$

n Number of resulting full words

count Number of partial words

nbits Number of rightmost bits of each partial word that contains useful data

If *count·nbits* is not a multiple of 64, the final partial word items will be left-justified, zero-filled.

SEE ALSO

UNPACK(3F)

NAME

PXFACCESS – Checks accessibility of a named file

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, iamode, ierror
CALL PXFACCESS(path, ilen, iamode, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFACCESS uses the access() function to check the accessibility of a named file.

The value of *iamode* indicates specific file permissions. These file permissions are checked against the current file permissions specified for the file in *path*. If the *iamode* permissions are allowed for the file in *path*, PXFACCESS returns a zero in *ierror*. Otherwise, it returns a nonzero value.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of a file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling access().

iamode An input integer variable containing the integer value of the symbolic constant for one or more of the following permissions: R_OK, W_OK, X_OK, or F_OK. An integer value for each of these symbolic constants is retrieved through the use of PXFCONST or IPXFCONST. The integer values may be combined through the use of a bitwise inclusive OR function.

ierror An output integer variable that will contain the status:

Zero	The requested access is permitted.
Nonzero	The requested access is not permitted.
	In addition to the errors returned by the access(2) system call, PXFACCESS may return the following errors:
EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH).
ENOMEM	If PXFACCESS is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, iamod, ierr
path = 'testfile'
iamod = 0
ilen = 0
ierr = 0
call pxfconst('R_OK',iamod,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfconst R_OK = ',ierr
else
    print *,'PASS: No error from pxfconst R_OK = '
endif
ierr = 0
call pxfaccess(path,ilen,iamod,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfaccess = ',ierr
else
    print *,'PASS: No error from pxfaccess = '
endif
end
```

NAME

PXFCHDIR – Change current directory to a specified directory

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, ierror
CALL PXFCHDIR(path, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFCHDIR uses the chdir() system call to change the current working directory to the specified directory.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the new directory.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling chdir().

ierror An output integer variable that will contain the status:

Zero	The current working directory was changed to the directory specified in *path*.
Nonzero	The change of directories was not made.
	In addition to the errors returned by the chdir(2) system call, PXFCHDIR may return the following errors:
EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH)

ENOMEM If PXFCHDIR is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, ierr
path = 'dir/testdir'
ilen = 0
call pxfchdir(path,ilen,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfchdir = ',ierr
else
    print *,'PASS: No error from pxfchdir = '
endif
end
```

NAME

PXFCHMOD – Sets file modes for a named file

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, imode, ierror
CALL PXFCHMOD(path, ilen, imode, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFCHMOD uses the chmod() function to set file modes for the named file.

The value of *imode* indicates specific file modes. chmod() changes the current file modes for the named file in *path* to the file mode specified in *imode*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path Input character variable or array element containing the name of a file.

ilen Input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling chmod().

imode Input integer variable containing the integer value of the symbolic constant for one or more of the following file modes:

USER	READ permissions bit: S_IRUSR
	WRITE permissions bit: S_IWUSR
	SEARCH/EXECUTE permissions bit: S_IXUSR
	Inclusive OR of READ/WRITE/EXECUTE: S_IRWXU

GROUP READ permissions bit: S_IRGRP
WRITE permissions bit: S_IWGRP
SEARCH/EXECUTE permissions bit: S_IXGRP
Inclusive OR of READ/WRITE/EXECUTE: S_IRWXG

OTHER READ permissions bit: S_IROTH
WRITE permissions bit: S_IWOTH
SEARCH/EXECUTE permissions bit: S_IXOTH
Inclusive OR of READ/WRITE/EXECUTE: S_IRWXO

SETID Set user ID on execution: S_ISUID
Set group ID on execution: S_ISGID

An integer value for each of these symbolic constants is retrieved through the use of PXFCONST or IPXFCONST. The integer values may be combined through the use of a bitwise inclusive OR function.

ierror Output integer variable that will contain the status:

Zero Requested file mode bits are set.

Nonzero Requested file mode bits are set.

In addition to the errors returned by the chmod(2) system call, PXFCHMOD may return the following errors:

EINVAL If *ilen* < 0 or *ilen* > LEN(PATH).

ENOMEM If PXFCHMOD is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, imod, ierr, imodr, imodw
path = 'testfile'
imod = 0
imodr = 0
imodw = 0
ilen = 0
call pxfconst('S_IRUSR',imodr,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfconst S_IRUSR = ',ierr
    stop
endif
call pxfconst('S_IROTH',imodw,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfconst S_IWUSR = ',ierr
    stop
endif
imod = IOR(imodr,imodw)
call pxfchmod(path,ilen,imod,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfchmod = ',ierr
else
    print *,'PASS: No error from pxfchmod'
endif
end
```

NAME

PXFCHOWN – Changes owner and group of a file

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, iowner, igroup, ierror
CALL PXFCHOWN(path, ilen, iowner, igroup, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFCHOWN uses the chown() function to change the owner and group of a file.

The value of *iowner* and *igroup* indicates the new values.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path Input character variable or array element containing the name of a file.

ilen Input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling chown().

iowner Input integer variable containing the integer value for the owner.

igroup Input integer variable containing the integer value for the group.

ierror Output integer variable that will contain the status:

Zero The group and owner of the file were changed.

Nonzero	PXFCHOWN did not change the group and owner.
	In addition to the errors returned by the chown(2) system call, PXFCHOWN may return the following errors:
EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH).
ENOMEM	If PXFCHOWN is unable to obtain memory to copy *path*.

EXAMPLES

```
subroutine test (iowner,igroup)
character*(12) path
integer ilen, iowner, igroup, ierr
path = 'testfile'
ilen = 0
call pxfchown(path,ilen,iowner,igroup,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfchown = ',ierr
else
    print *,'PASS: No error from pxfchown = '
endif
end
```

NAME

PXFCHROOT – Changes root directory to specified directory

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, ierror
CALL PXFCHROOT(path, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFCHROOT uses the chroot() system call to change the root directory to the specified directory.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path Input character variable or array element containing the name of a file.

ilen Input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling chroot().

ierror Output integer variable that will contain the status:

Zero	Root directory was changed to the directory specified by *path*.
Nonzero	Root directory was not changed.
	In addition to the errors returned by the chroot(2) system call, PXFCHROOT may return the following errors:
EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH).
ENOMEM	If PXFCHROOT is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, ierr
path = '/dir/test'
ilen = 0
call pxfchroot(path,ilen,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfchroot = ',ierr
else
    print *,'PASS: No error from pxfchroot'
endif
end
```

NAME

PXFCONST – Returns value associated with symbolic constants

SYNOPSIS

```
CHARACTER*(n) constname
INTEGER ival, ierror
LOGICAL PXFISCONST, l

i = IPXFCONST (constname)
l = PXFISCONST (constname)
CALL PXFCONST (constnam, ival, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These functions provide a way for the Fortran programmer to obtain the value of some symbolic constants defined in system header files.

IPXFCONST() provides an integer return value but no error checking. If the argument passed corresponds to one of the defined constants shown below, the return value is the integer value associated with the constant; if the argument is not a defined constant, the return value is meaningless. PXFISCONST() confirms whether the argument is a valid constant. PXFISCONST() returns .TRUE only if IPXFCONST() would return a valid value for the same *constname*.

Upon successful completion, the subroutine PXFCONST() returns in *ival* the integer value associated with the constant described by *constname*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of arguments for these routines:

constname An input character variable that represents the name of a constant. *constname* is case-sensitive, and trailing blanks in the argument are ignored.

The following are valid values for *constname*. The second column contains the system header file where the symbolic constant is defined, or the standard where it is defined.

```
'F_GETLK'        <fcntl.h>
'F_SETLK'        <fcntl.h>
'F_SETLKW'       <fcntl.h>
'F_RDLCK'        <fcntl.h>
'F_WRLCK'        <fcntl.h>
'F_UNLCK'        <fcntl.h>
'F_DUPFD'        <fcntl.h>
'F_GETFD'        <fcntl.h>
'F_SETFD'        <fcntl.h>
'F_GETFL'        <fcntl.h>
'F_SETFL'        <fcntl.h>
'F_SETSB'        <fcntl.h>
'F_SETALF'       <fcntl.h>
'F_CLRALF'       <fcntl.h>
'O_RDONLY'       <fcntl.h>
'O_WRONLY'       <fcntl.h>
'O_RDWR'         <fcntl.h>
'O_ACCMODE'      <fcntl.h>
'O_NDELAY'       <fcntl.h>
'O_APPEND'       <fcntl.h>
'O_SYNC'         <fcntl.h>
'O_NONBLOCK'     <fcntl.h>
'O_RAW'          <fcntl.h>
'O_SSD'          <fcntl.h>
'O_CREAT'        <fcntl.h>
'O_TRUNC'        <fcntl.h>
'O_EXCL'         <fcntl.h>
'O_NOCTTY'       <fcntl.h>
```

```
'O_BIG'           <fcntl.h>
'O_PLACE'         <fcntl.h>
'O_RESTART'       <fcntl.h>
'O_ASYNC'         <fcntl.h>
'O_PTYIGN'        <fcntl.h>
'O_SFSXOP'        <fcntl.h>
'O_LDRAW'         <fcntl.h>
'O_WELLFORMED'    <fcntl.h>
'O_SFS_DEFER_TM' <fcntl.h>

'S_ALF_NOGROW'    <sys/stat.h>
'S_ALF_PARTR'     <sys/stat.h>

'SEEK_SET'        <stdio.h>
'SEEK_CUR'        <stdio.h>
'SEEK_END'        <stdio.h>

'STDIN_FILENO'    Posix.9
'STDOUT_FILENO'   Posix.9
'STDERR_FILENO'   Posix.9
```

Posix.9 specific errors:

```
'ENONAME'         liberrno.h
'ENOHANDLE'       liberrno.h
```

Errnos 1–98 from `<errno.h>`, for example: `'EPERM'`

Additional values for *constname* are described in descriptions of other PXF routines such as `PXFACCESS`, `PXFCHMOD`, `PXFCREAT`, `PXFOPEN`, and so on.

ival An output integer variable. The value associated with the constant.

ierror An output integer variable. It contains the exit status.

EXIT STATUS

Upon successful completion, the argument *ierror* is set to 0. If any of the following conditions occur, PXFCONST() sets *ierror* to the corresponding value.

[ENONAME]	Invalid constant name
[ENOMEM]	PXFCONST() could not allocate the memory required

NAME

PXFCREAT – Creates a new file or rewrites an existing file

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, imode, ifildes, ierror
CALL PXFCREAT(path, ilen, imode, ifildes, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFCREAT uses the creat() system call to create a new file or rewrite an existing file.

The call is similar to PXFOPEN with an IOPENFLAG argument of O_WRONLY, O_CREAT, and O_TRUNC.

The value of *imode* indicates specific file modes. If the file exists, *imode* is ignored. The mode values are used when *path* is a new file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of a file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling creat().

imode An input integer variable containing the integer value of the symbolic constant for one or more of the following file modes:

USER	READ permissions bit: S_IRUSR WRITE permissions bit: S_IWUSR SEARCH/EXECUTE permissions bit: S_IXUSR Inclusive OR of READ/WRITE/EXECUTE: S_IRWXU
GROUP	READ permissions bit: S_IRGRP WRITE permissions bit: S_IWGRP SEARCH/EXECUTE permissions bit: S_IXGRP Inclusive OR of READ/WRITE/EXECUTE: S_IRWXG
OTHER	READ permissions bit: S_IROTH WRITE permissions bit: S_IWOTH SEARCH/EXECUTE permissions bit: S_IXOTH Inclusive OR of READ/WRITE/EXECUTE: S_IRWXO
SETID	Set user ID on execution: S_ISUID Set group ID on execution: S_ISGID

An integer value for each of these symbolic constants is retrieved through the use of PXFCONST or IPXFCONST. The integer values may be combined through the use of a bitwise inclusive OR function.

ifildes An output integer variable containing the file descriptor returned by creat().

ierror An output integer variable that will contain the status:

Zero	The file is created or rewritten.
Nonzero	PXFCREAT is not successful.

In addition to the errors returned by the creat(2) system call, PXFCREAT may return the following errors:

EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH).
ENOMEM	If PXFCREAT is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, imod, ifilde, ierr
integer imodru, imodwu, imodwg, imodrg
integer ierr1, ierr2, ierr3, ierr4
path = 'testfile'
imod = 0
ilen = 0
call pxfconst('S_IRUSR',imodru,ierr1)
call pxfconst('S_IWUSR',imodwu,ierr2)
call pxfconst('S_IRGRP',imodrg,ierr3)
call pxfconst('S_IWGRP',imodwg,ierr4)
imod = IOR((IOR(imodru,imodwu)),(IOR(imodrg,imodwg)))
call pxfcreat(path,ilen,imod,ifilde,ierr)
if (ierr.ne.0) then
   print *,'FAIL: error from pxfcreat = ',ierr
else
   print *,'PASS: No error from pxfcreat = '
endif
end
```

NAME

PXFFCNTL – Provides a subset of fcntl(2) functionality, except the third argument is always an integer

SYNOPSIS

```
INTEGER IFILDES, ICMDP, IARGIN, IARGOUT, IERROR
CALL PXFFCNTL (ifildes, icmd, iargin, iargout, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The subroutine PXFFNCTL() provides a subset of the functionality of fcntl(2), except that the third argument is always an integer.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The arguments for PXFFNCTL() are:

ifildes An open file descriptor. *ifildes* is an input integer variable.

icmd An input integer variable. Specifies the action for fcntl to perform, as described on the fcntl(2) man page. The constant values for use in specifying *icmd* are available through calls to PXFCONST(). The following values are currently supported for *icmd*:

F_CLRALF

F_DUPFD

F_GETFD

F_GETFL

F_GETLK

F_SETALF

F_SETFD

F_SETFL

F_SETSB

F_SETLK

F_SETLKW

iargin An input integer variable. As described below, *iargin* may also be an output variable. *iargin* can be a handle for an instance of the flock structure or an integer, depending on the argument ICMD under the conditions defined in POSIX.1.

If *icmd* is F_GETLK, F_SETLK, or F_SETLKW, then *iargin* is a handle for an instance of the flock structure. This handle should be created by a call to the PXFSTRUCTCREATE() subroutine, with the string *flock* given as the STRUCTNAME argument. The components are then accessed with the subroutines PXFINTSET() and PXFINTGET(), as shown following:

Components for flock structure:

Posix.1 component	COMPNAM	Structure procedures used to access
l_type	l_type	PXFINTGET, PXFINTSET
l_whence	l_whence	PXFINTGET, PXFINTSET
l_start	l_start	PXFINTGET, PXFINTSET
l_len	l_len	PXFINTGET, PXFINTSET
l_pid	l_pid	PXFINTGET, PXFINTSET

For *icmd* F_GETLK, the retrieved information overwrites the information described in the handle used as the *iargin* argument.

iargout An output integer variable. The value returned in *iargout* depends on the *icmd* argument. See fcntl() for more information.

ierror An output integer variable. It contains the exit status.

EXIT STATUS

Upon successful completion of PXFFCNTL(), the argument *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

[EINVAL]	ICMD is not valid.
[EINVAL]	ICMD is F_GETLK, F_SETLK, or F_SETLKW and *iargin* is not a valid handle.
[errno]	The fcntl(2) system call failed.

NAME

PXFFILENO – Returns the file descriptor for a specified unit

SYNOPSIS

```
INTEGER IUNIT, IFILDES, IERROR
CALL PXFFILENO (iunit, ifildes, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The PXFFILENO() subroutine returns in *ifildes* the file descriptor to which the unit identified by *iunit* is connected.

Processing of some Fortran file types includes library buffering or the addition of control words to the data written. Users should be aware of this when attempting to make use of the file descriptor associated with a Fortran unit. Some Fortran units may not be connected to a file descriptor. For example, a file assigned the attributes −F *mr.scr.novfl* is not connected to a file descriptor.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of arguments to PXFFILENO:

iunit An input integer variable that contains the Fortran unit number.

ifildes An output integer variable containing the file descriptor of the file identified by *iunit*.

ierror An output integer variable containing the status.

EXIT STATUS

Upon successful completion of PXFFILENO(), *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

[EINVAL] *iunit* is not an open unit

[EBADF] *iunit* is not connected with a file descriptor

NAME

PXFGETARG – Returns command-line argument

SYNOPSIS

```
CHARACTER*n buf
INTEGER m, ilen, ierror
CALL PXFGETARG(m, buf, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

The subroutine PXFGETARG examines the command line used to invoke the executing program and returns the *m*th command-line argument in *buf*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine has the following arguments:

m An input integer variable or array element containing the number of the command-line argument to return in *buf*. If *m* is zero, the command name is returned in *buf*.

buf An output character variable or array element containing the *m*th command-line argument.

ilen An output integer variable containing the significant length of *buf* in characters (that is, the actual length of the argument in *buf*).

ierror An output integer variable containing the status:

EINVAL is returned if *m* is out of range.

ETRUNC is returned if the declared length of *buf* is insufficient to contain the string to be returned. The value of the command-line argument is truncated to fit in *value*, and *ilen* is set to the full length of *buf*.

A 0 is returned if PXFGETARG is successful (if a value is returned in *buf*).

SEE ALSO

IPXFARGC(3F)

NAME

PXFGETENV – Returns value for environment name

SYNOPSIS

```
CHARACTER*n name, value
INTEGER lenname, lenval, ierror
CALL PXFGETENV (name, lenname, value, lenval, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

The subroutine PXFGETENV uses the getenv() function to search the environment list for a *name* in a string of the form *name=value*.

If *name* matches a name in the list, the character representation of *value* is stored in the *value* character argument and the number of characters in *value* is stored in *lenval*. If the length of the value to be placed in *value* is larger than the declared length of *value*, the value string is truncated on the right and stored in *value*. The nontruncated length is stored in *lenval* and *ierror* is set to *etrunc*. If the length of the value is shorter than the declared size of *value*, the value string is stored with left justification and filled with blanks on the right. *lenval* is set to the shorter length of the value string.

If *name* is found but has no value, blanks are stored in *value* and *lenval* is set to zero. If *name* cannot be found, *einval* is returned in *ierror*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

name An input character variable or array element containing the name of an environment variable.

lenname An input integer variable containing the length of *name* in characters. If *lenname* is zero, the trailing blanks are removed. The declared length of the input name is decremented by the number of blanks removed. If *lenname* is zero and *name* is all blanks, the input name is a null string.

value An output character variable or array element containing the value of the environment variable *name*.

lenval An output integer variable containing the length of *value* in characters. If *name* is found but has no value, *lenval* is zero and *value* contains all blanks to indicate a NULL string. If the value representation is truncated to be stored in *value*, *lenval* contains the nontruncated length of value. If the value representation is shorter than the length of *value*, *lenval* contains the shorter length.

ierror An output integer variable containing the status:

EINVAL is returned if *name* is not in the environment list.

ETRUNC is returned if the declared length of *value* is insufficient to contain the string to be returned. The value of *name* is truncated to fit in *value*, and *lenval* contains the original length of the value of *name* before truncation.

A 0 if getenv is successful (if *name* is found).

EXAMPLES

In the example, PXFGETENV will search for a string containing SHELL=*value*.

```
      program testpxf
      character*24 namea, nameb
      integer lena, lenb, ier
c set input arguments
      ier = 0
      lena=0
      lenb=0
      namea='SHELL'
      nameb=' '
      CALL PXFGETENV (namea, lena, nameb, lenb, ier)
      print *,'TEST results:'
c print input arguments
      print *,'namea=-',namea,'-'
      print *,'lena=',lena
c print output arguments
      print *,'nameb=-',nameb,'-'
      print *,'lenb=',lenb
      print *,'ier=',ier
      end
```

If the string is found, it may return:

```
TEST results:
namea=-SHELL
lena=0
nameb=-/bin/csh
lenb=8
ier=0
```

NAME

PXFINTGET – Allows values stored in individual components of a structure to be extracted and used

SYNOPSIS

```
INTEGER JHANDLE, VALUE, IERROR
CHARACTER*(n) compnam
CALL PXFINTGET (jhandle, compnam, value, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The subroutine PXFINTGET() allows values stored in individual components of a structure to be extracted and used.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are arguments to PXFINTGET:

jhandle An input integer that references a structure. *jhandle* should have been created with PXFSTRUCTCREATE().

compnam An input character variable. *compnam* is the name of a component of the structure. *compnam* is case-sensitive.

value An output integer variable. Upon successful completion, *value* is set to the value stored in the component of *jhandle* referenced by *compnam*. The structures and components that can be accessed through PXFINTGET() are described on the man page for the related routine.

ierror An output integer variable. It contains the exit status.

EXIT STATUS

Upon successful completion of `PXFINTGET()`, the argument *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

`[ENONAME]` Component name is not defined for this structure

NAME

PXFINTSET – Allows components of a structure to be set or modified

SYNOPSIS

```
INTEGER JHANDLE, VALUE, IERROR
CHARACTER*(n) compnam
CALL PXFINTSET (jhandle, compnam, value, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The subroutine PXFINTSET() allows components of a structure to be set or modified.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The arguments for PXFINTSET are:

jhandle An input integer variable. *jhandle* references a structure. *jhandle* should have been created by PXFSTRUCTNAME().

compnam An input character variable. *compnam* is the name of a component of the structure. *compnam* is case-sensitive.

value An input integer variable. *value* is the value to be stored in the component of *jhandle* referenced by *compnam*. The structures and components that can be accessed through PXFINTSET() are described on the man page for the related routine.

ierror An output integer variable.

EXIT STATUS

Upon successful completion of PXFINTSET(), the argument *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

[ENONAME] Component name is not defined for this structure

NAME

PXFLINK – Creates a link to a file

SYNOPSIS

```
CHARACTER*n existf, newf
INTEGER lenexist, lennew, ierror
CALL PXFLINK(existf, lenexist, newf, lennew, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFLINK uses the link() function to link an existing file to a new file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

existf An input character variable or array element containing the name of an existing file.

lenexist An input integer variable containing the length of *existf* in characters. If *lenexist* is zero, all trailing blanks are removed before calling link().

newf An input character variable or array element containing the name of a new file.

lennew An input integer variable containing the length of *newf* in characters. If *lennew* is zero, all trailing blanks are removed before calling link().

ierror An output integer variable that will contain the status:

Zero The link was successful.

Nonzero The link was not completed.

In addition to the errors returned by the `link`(2) system call, `PXFLINK` may return the following errors:

`EINVAL` If *lenexist* < 0 or *lenexist* > `LEN(EXISTF)` or *lennew* < 0 or *lennew* > `LEN(NEWF)`.

`ENOMEM` If `PXFLINK` is unable to obtain memory to copy *existf* or *newf*.

EXAMPLES

```
program test
character*(12) filea, fileb
integer lenfila, lenfilb,ierr
filea = 'existfile'
fileb = 'newfile'
lenfila = 0
lenfilb = 0
call pxflink(filea,lenfila,fileb,lenfilb,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxflink = ',ierr
else
    print *,'PASS: No error from pxflink = '
endif
end
```

NAME

PXFOPEN – Provides a Fortran interface to the open(2) system call

SYNOPSIS

```
INTEGER ilen, iopenflag, imode, ifildes, ierror
CHARACTER*n path
CALL PXFOPEN(path, ilen, iopenflag, imode, ifildes, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFOPEN provides a subset of the functionality of the open(2) system call.

path identifies the file to be opened. If PXFOPEN successfully opens the file, the file descriptor is returned in the argument *ifildes*, and *ierror* is set to zero. If the file cannot be opened, *ierror* is set to the error value.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of arguments for this routine:

path An input character variable or array element containing the name of the file to be opened.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, the trailing blanks are removed.

iopenflag An input integer variable containing the status flags. See the open(2) man page for more detail. The value for this variable may be obtained through the use of PXFCONST or IPXFCONST. The following values are currently supported for *iopenflag*:

- O_RDONLY

- O_WRONLY
- O_RDWR
- O_RAW
- O_LDRAW
- O_NDELAY
- O_NONBLOCK
- O_NOCTTY
- O_BIG
- O_APPEND
- O_CREAT
- O_TRUNC
- O_EXCL
- O_PLACE
- O_RESTART
- O_SSD
- O_SYNC
- O_WELLFORMED

The integer values may be combined through the use of a bitwise inclusive OR function.

imode An input integer variable, denoting the file access permission. The value for this variable may be retrieved through the use of PXFCONST or IPXFCONST. The following values are currently supported for *imode*:

USER	READ permissions bit: S_IRUSR WRITE permissions bit: S_IWUSR SEARCH/EXECUTE permissions bit: S_IXUSR Inclusive OR of READ/WRITE/EXECUTE: S_IRWXU
GROUP	READ permissions bit: S_IRGRP WRITE permissions bit: S_IWGRP SEARCH/EXECUTE permissions bit: S_IXGRP Inclusive OR of READ/WRITE/EXECUTE: S_IRWXG

	OTHER	READ permissions bit: S_IROTH WRITE permissions bit: S_IWOTH SEARCH/EXECUTE permissions bit: S_IXOTH Inclusive OR of READ/WRITE/EXECUTE: S_IRWXO
	SETID	Set user ID on execution: S_ISUID Set group ID on execution: S_ISGID
		The integer values may be combined through the use of a bitwise inclusive OR function.

ifildes An output integer variable. If the open() system call was successful, *ifildes* will be set to the file descriptor.

ierror An output integer variable containing the status:

0	If PXFOPEN was successful (the open(2) succeeded)
errno	If the open(2) system call failed
EINVAL	If *ilen* is < 0 or *ilen* > LEN(PATH)

NOTES

PXFOPEN does not provide a way to specify the *cbits* or *cblks* parameters to open(2).

NAME

PXFRENAME – Renames a file

SYNOPSIS

```
CHARACTER*n oldnm, newnm
INTEGER lenold, lennew, ierror
CALL PXFRENAME(oldnm, lenold, newnm, lennew, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFRENAME uses the rename() function to change the name of a file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

oldnm An input character variable or array element containing the current file name.

lenold An input integer variable containing the length of *oldnm* in characters. If *lenold* is zero, all trailing blanks are removed before calling rename().

newnm An input character variable or array element containing the new name of the file.

lennew An input integer variable containing the length of *newnm* in characters. If *lennew* is zero, all trailing blanks are removed before calling rename().

ierror An output integer variable that will contain the status:

Zero	The renaming of the file was successful.
Nonzero	The renaming of the file was not completed.

In addition to the errors returned by rename(2) system call, PXFRENAME may return the following errors:

EINVAL If *lenold* < 0 or *lenold* > LEN(OLD) or *lennew* < 0 or *lennew* > LEN(NEWF).

ENOMEM If PXFRENAME is unable to obtain memory to copy *oldnm* or *newnm*.

EXAMPLES

```
program test
character*(12) filea, fileb
integer lenfila, lenfilb,ierr
filea = 'oldname'
fileb = 'newname'
lenfila = 0
lenfilb = 0
call pxfrename(filea,lenfila,fileb,lenfilb,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfrename = ',ierr
else
    print *,'PASS: No error from pxfrename = '
endif
end
```

NAME

PXFRMDIR – Removes a directory entry

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, ierror
CALL PXFRMDIR(path, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFRMDIR uses the rmdir() function to remove a directory entry for the named file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of a file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling rmdir().

ierror An output integer variable that will contain the status:

Zero The named file was removed.

In addition to the errors returned by the rmdir(2) system call, PXFRMDIR may return the following errors:

EINVAL If *ilen* < 0 or *ilen* > LEN(PATH).

ENOMEM If PXFRMDIR is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, ierr
path = 'testfile'
ilen = 0
call pxfrmdir(path,ilen,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfrmdir = ',ierr
else
    print *,'PASS: No error from pxfrmdir = '
endif
end
```

NAME

PXFSTAT – Retrieves the file status

SYNOPSIS

```
INTEGER jstat, ilen, ierror
CALL PXFSTAT(path, ilen, jstat, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFSTAT uses the stat() system call to get the file status.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of a file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling stat().

jstat An input integer variable or array element containing a handle for a stat structure. This handle should have been created by a call to the PXFSTRUCTCREATE(3F) routine.

ierror An output integer variable that will contain the status:

Zero	PXFSTAT returned the status information.
Nonzero	PXFSTAT was unable to return the status.
	In addition to the errors returned by the stat(2) system call, PXFSTAT may return the following errors:
EINVAL	If *ilen* < 0 or *ilen* > LEN(PATH).
ENOMEM	If PXFSTAT is unable to obtain memory to copy *path*.

The stat structure contains the following components:

- st_mode: File mode.
- st_ino: File serial number.
- st_dev: ID of devide containing the file.
- st_nlink: Number of links.
- st_uid: User id of the owner of the file.
- st_gid: Group id of the owner of the file.
- st_size: File size in bytes for regular files. Unspecified for other files.
- st_atime: Last time that data within the file was accessed.
- st_mtime: Last time of that data in the file was modified.
- st_ctime: Last time that file status was changed.

EXAMPLES

```
program test
character*10 path
integer ilen, jstat, ierr,imode, istino
path = 'stt.f'
call pxfstructcreate('stat',jstat,ierr)
   print *,'structcreate error = ',ierr
ilen=0
call pxfstat(path, ilen, jstat, ierr)
if (ierr.ne.0) then
   print *,'FAIL: error from pxfstat = ',ierr
else
   print *,'PASS: No error from pxfstat = '
endif
call pxfintget(jstat,'st_ino',istino,ierr)
call pxfintget(jstat,'mode',imode,ierr)
print *,'st_ino = ',istino
print *,'mode = ',imode
call pxfstructfree(stat,ierr)
end
```

NAME

PXFSTRGET – Allows values stored in individual components of a structure to be extracted and used

SYNOPSIS

```
INTEGER JHANDLE, ILEN, IERROR
CHARACTER*(n) compnam, value
CALL PXFSTRGET (jhandle, compnam, value, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFSTRGET() allows character values stored in individual components of a structure to be extracted and used.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are arguments to PXFSTRGET:

jhandle An input integer that references a structure. *jhandle* should have been created with PXFSTRUCTCREATE().

compnam An input character variable. *compnam* is the name of a component of the structure. *compnam* is case-sensitive.

value An output character variable. Upon successful completion, *value* is set to the value stored in the component of *jhandle* referenced by *compnam*. If the length of the data being stored is less than the declared length of *value*, *value* is padded with blanks.

The structures and components that may be accessed through PXFSTRGET() are described on the man page for the related PXF routine. For example, the PXUNAME man page describes the components for the *utsname* structure.

ilen An output integer variable. *ilen* is set to the actual length of the data assigned to value. If the length of value is insufficient to contain the data being returned, the data is truncated, and *ilen* contains the original length of the data before truncation. In this case, *ierror* is set to ETRUNC.

ierror An output integer variable. Upon successful completion of PXFSTRGET, ierror is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

[ENONAME] Component name is not defined for the specified structure.

[ETRUNC] The actual length of the data to be copied to *value* was longer than the declared length of *value*.

SEE ALSO

PXFUNAME(3F)

NAME

PXFSTRSET – Allows values stored in individual components of a structure to be set

SYNOPSIS

```
INTEGER JHANDLE, ILEN, IERROR
CHARACTER*(n) compnam, value
CALL PXFSTRSET (jhandle, compnam, value, ilen, ierror)
```

IMPLEMENTATION

All Cray Research system

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFSTRSET() allows character values stored in individual components of a structure to be set.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are arguments to PXFSTRSET:

jhandle An input integer that references a structure. *jhandle* should have been created with PXFSTRUCTCREATE().

compnam An input character variable. *compnam* is the name of a component of the structure. *compnam* is case-sensitive.

value An input character variable. Upon successful completion, the component of *jhandle* referenced by *compnam* is set to *value*. The structures and components that can be accessed through PXFSTRSET() are described on the man page for the related PXF routine.

ilen An input integer variable. *ilen* is the intended length of value. If *ilen* is zero, then trailing blanks in value are stripped and ignored.

ierror An output integer variable. Upon successful completion of PXFSTRSET, *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

[ENONAME] Component name is not defined for the specified structure.

NAME

PXFSTRUCTCREATE – Creates an instance of the desired structure and returns a nonzero handle in the argument *jhandle*

SYNOPSIS

```
INTEGER JHANDLE, IERROR
CHARACTER*(n) structname
CALL PXFSTRUCTCREATE (structname, jhandle, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The subroutine PXFSTRUCTCREATE() creates an instance of the desired structure and returns a nonzero handle in the argument *jhandle*. All further references to this instance of the structure are through this handle. The initial values of components within the new instance of the structure are undefined.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are arguments for PXFSTRUCTCREATE:

structname An input character variable. *structname* specifies which type of structure should be created. Values for *structname* that are currently recognized are shown in the following table. *structname* must be in lowercase letters; trailing blanks are ignored.

Values for *structname*:

STRUCTNAME	Structure name	Header file containing definition
FLOCK	flock	<fcntl.h>
UTIMBUF	utimbuf	<utime.h>
UTSNAME	utsname	<sys/utsname.h>
STAT	stat	<stat.h>

jhandle An output integer variable. The structure handle is returned in *jhandle*.

ierror An output integer variable.

EXIT STATUS

Upon successful completion of PXFSTRUCTCREATE(), the argument *ierror* is set to 0. If any of the following conditions occur, PXFSTRUCTCREATE() sets the argument to the corresponding value:

[ENONAME] Component name is not defined for this structure

[ENOHANDLE] Instance of the structure could not be created

EXAMPLES

```
      program test
      integer junam, ierr, ilen
      character*15 sname
*  Create STRUCTURE to be used by uname()
      call pxfstructcreate('utsname',junam,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfstructcreate = ',ierr
      endif
*  Fill STRUCTURE through uname()
      call pxfuname(junam,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfuname = ',ierr
      endif
      ilen = 0
*  Retrieve component sysname from STRUCTURE
      call pxfstrget(junam,'sysname',sname,ilen,ierr)
      print *, 'sysname=',sname
*  Free STRUCTURE
      call pxfstructfree(junam,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfstructfree = ',ierr
      endif
      end
```

NAME

PXFSTRUCTFREE – Deletes the instance of the structure referenced by *jhandle*

SYNOPSIS

```
INTEGER JHANDLE, IERROR
CALL PXFSTRUCTFREE (jhandle, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The subroutine PXFSTRUCTFREE() deletes the instance of the structure referenced by *jhandle*. This structure should have been created by PXFSTRUCTCREATE().

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are arguments to PXFSTRUCTCREATE():

jhandle An input integer variable. *jhandle* is a handle for a structure.

ierror An output integer variable. Upon successful completion of PXFSTRUCTFREE(), *ierror* is set to 0.

EXAMPLES

```
      program test
      integer jstat, ierr, ilen, imode, istino
      character*10 path
*  Create STRUCTURE to be used by stat()
      call pxfstructcreate('stat',jstat,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfstructcreate = ',ierr
      endif
*  Fill STRUCTURE through stat()
      ilen = 0
      call pxfstat(path, ilen, jstat,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfstat = ',ierr
      endif
*  Retrieve components st_ino and mode from STRUCTURE
      call pxfintget(jstat,'st_ino',isino,ierr)
      call pxfintget(jstat,'mode',imode,ierr)
      print *, 'st_ino =',stino
      print *, 'mode =',mode
*  Free STRUCTURE
      call pxfstructfree(jstat,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: error from pxfstructfree = ',ierr
      endif
      end
```

NAME

PXFUNAME – Retrieves the operating system name

SYNOPSIS

```
INTEGER junam, ierror
CALL  PXFUNAME(junam, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFUNAME uses the uname() system call to get the components of the operating system name.

The components of the utsname structure are:

- SYSNAME: Name of the operating system
- NODENAME: Name of node in the operating system
- RELEASE: Current release level of the operating system
- VERSION: Current version level of the release
- MACHINE: Name of hardware type currently executing the program

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

junam An input integer variable or array element containing a handle for a utsname structure. This should have been created by a call to the PXFSTRUCTCREATE(3F) routine.

ierror An output integer variable that will contain the status:

Zero PXFUNAME returned the structure successfully.

Nonzero PXFUNAME was unable to return the structure.

EXAMPLES

```
program test
integer junam, ierr, ilen
character*15 sname, nname, rel, vers, mach
call pxfstructcreate('utsname',junam,ierr)
call pxfuname(junam,ierr)
IF (ierr.ne.0) then
   print *,'FAIL: error from pxfuname = ',ierr
else
   print *,'PASS: No error from pxfuname = '
endif
ilen = 0
call pxfstrget(junam,'sysname',sname,ilen,ierr)
ilen = 0
call pxfstrget(junam,'nodename',nname,ilen,ierr)
ilen = 0
call pxfstrget(junam,'release',rel,ilen,ierr)
ilen = 0
call pxfstrget(junam,'version',vers,ilen,ierr)
ilen = 0
call pxfstrget(junam,'machine',mach,ilen,ierr)
print *, 'sysname=',sname
print *, 'nodename=',nname
print *, 'release=',rel
print *, 'version=',vers
print *, 'machine=',mach
call pxfstructfree(junam,ierr)
```

NAME

PXFUNLINK – Removes a directory entry

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, ierror
CALL PXFUNLINK(path, ilen, ierror)
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFUNLINK uses the unlink() function to remove a directory entry for the named file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of a file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling unlink().

ierror An output integer variable that will contain the status:

Zero The named file was removed.

In addition to the errors returned by the unlink(2) system call, PXFUNLINK may return the following errors:

EINVAL If *ilen* < 0 or *ilen* > LEN(PATH).

ENOMEM If PXFUNLINK is unable to obtain memory to copy *path*.

EXAMPLES

```
program test
character*(12) path
integer ilen, ierr
path = 'testfile'
ilen = 0
call pxfunlink(path,ilen,ierr)
if (ierr.ne.0) then
    print *,'FAIL: error from pxfunlink = ',ierr
else
    print *,'PASS: No error from pxfunlink = '
endif
end
```

NAME

PXFUTIME – Sets access and modification times of a file

SYNOPSIS

```
CHARACTER*n path
INTEGER ilen, jutimbuf, ierror
CALL PXFUTIME(path, ilen, jutimbuf, ierror)
```

IMPLEMENTATION

Cray PVP systems and Cray MPP systems only

STANDARDS

IEEE standard interface for FORTRAN 77

DESCRIPTION

The subroutine PXFUTIME provides the functionality of the utime(2) system call.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex, and logical arguments.

The following is a list of valid arguments for this routine:

path An input character variable or array element containing the name of the file.

ilen An input integer variable containing the length of *path* in characters. If *ilen* is zero, all trailing blanks are removed before calling utime().

jutimbuf An input integer variable. It is a handle for a structure of type utimbuf. The handle must be created by a call to the PXFSTRUCTCREATE(3F) subroutine prior to the call to PXFUTIME. The names of the components of the utimbuf structure are actime and modtime. These components can be accessed through the PXFINTSET subroutine. The functionality obtained in utime() by passing a NULL can be obtained in PXFUTIME by passing a handle argument with a value of zero.

ierror An output integer variable that will contain the status:

Zero The call to utime was successful.

Nonzero The call to `utime` was not completed.

In addition to the errors returned by the `utime`(2) system call, `PXFUTIME` may return the following errors:

`EINVAL` If *ilen* < 0 or *ilen* > `LEN(PATH)`.

`ENOMEM` If `PXFUTIME` is unable to obtain memory to copy *path*.

EXAMPLES

```
      program test

      character*12 filea, fileb
      integer ilenfila, ilenfilb, jutimbuf, ierr

! create file
      open(file='existfile', unit=9, status='NEW')
      write(9,*) 'HI', 1.2, 11, 'GOODBYE'
      endfile 9
      close 9

      filea = 'existfile'
      ilenfila = 0
      ilenfilb = 0

! Set file access and modification time to current time
      call pxfutime(filea,ilenfila,0,ierr)
      if (ierr.ne.0) then
         print *,'FAIL: pxfutime'
         print *,'nonzero status on existing file = ',ierr
      else
         print *,'PASS: pxfutime'
         print *,'zero status on existing file'
      endif

      end
```

The output of this test on a PVP system is:

```
PASS: pxfutime
zero status on existing file
```

NAME

SECONDR – Returns elapsed wall-clock time in seconds

SYNOPSIS

time=SECONDR( )

Cray PVP systems:

CALL SECONDR(*time*)

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

SECONDR can be called several times during the execution of a program to determine the amount of elapsed wall-clock time.

time Real-time clock values in floating-point seconds. If the value of the real-time clock exceeds 48 bits, some accuracy may be lost during the conversion to float.

NOTES

On Cray MPP systems, the real-time clocks on processing elements are not synchronized with each other. Therefore, a SECONDR() value should not be compared to a previous SECONDR() taken on a different processing element.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

EXAMPLES

This example calculates the elapsed wall-clock time from the first call to SECONDR until the second call to SECONDR.

```
BEFORE = SECONDR( )
CALL DOWORK ( )
AFTER = SECONDR( )
WALTIM = AFTER - BEFORE
```

SEE ALSO

RTC(3I), SECOND(3F), TIMEF(3F), TSECND(3F)

NAME

SYSCLOCK – Returns real-time clock value and number of wraps

SYNOPSIS

```
INTEGER ICOUNT, IWRAP
CALL SYSCLOCK (icount, iwrap)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

SYSCLOCK returns the same real-time clock count returned by the *count* argument to the Fortran 90 SYSTEM_CLOCK intrinsic subroutine.

On 32-bit systems, the real-time clock value for SYSTEM_CLOCK reaches *count_max* and wraps around to 0 occasionally. The number of times that the clock has wrapped is returned in *iwrap*. By using *iwrap*, you can reliably time intervals during which the clock may have wrapped one or more times.

Use the *count_rate* argument on the SYSTEM_CLOCK intrinsic subroutine to determine the clock rate for SYSCLOCK.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

RETURN VALUES

iwrap always returns 0 on 64-bit systems.

SEE ALSO

IRTC(3I), SYSTEM_CLOCK(3I), TIMEF(3F)

NAME

TIMEF – Returns elapsed wall-clock time since the call to TIMEF

SYNOPSIS

timef=TIMEF()

Cray PVP systems:
CALL TIMEF(*timef*)

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

TIMEF returns the elapsed wall-clock time since a call to TIMEF.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

timef Elapsed wall-clock time (a real number in milliseconds) since the initial call to TIMEF.

Type real. The initial call to TIMEF returns 0.0.

result Same as *timef*.

NOTES

If TIMEF calculates a negative value for the elapsed wall-clock time, it resets the initial value of the clock to the current value of the
lock and returns 0.0.

NAME

TRIMLEN – Returns the length of a character argument without counting trailing blanks

SYNOPSIS

```
INTEGER trimlen
intlen  =  TRIMLEN(string)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TRIMLEN is an integer function that returns the length of a character argument without counting the trailing blanks. The function must be declared as type integer in the calling routine. The *string* argument must be type character.

This function is intended for use as part of the substring notation. Examples 2 and 3 show the TRIMLEN function used in this manner. The value of each of the parts of the substring notation must be as follows:

$$1 \leq leftmostpos \leq rightmostpos \leq stringlength$$

leftmostpos is the leftmost character position in the substring, *rightmostpos* is the rightmost character position in the substring, and *stringlength* is the declared length of the character entity. TRIMLEN returns a value of 1 for a string of all blanks.

EXAMPLES

Example 1: A program using the function TRIMLEN could do the following:

```
      INTEGER TRIMLEN
      CHARACTER*80 STRING
      STRING = ' '
      STRING(20:47) = 'TEST TRIMLEN LENGTH RETURNED'
      INTLEN = TRIMLEN(STRING)
      WRITE(6,1) INTLEN, STRING(1:INTLEN)
    1 FORMAT(' LENGTH=',I5,' STRING=',A,'-DONE')
      PRINT 2,'12345678901234567890123456789012345678901234567890'
    2 FORMAT(21X,A)
      END
```

The output of the program is as follows:

```
LENGTH=    47 STRING=                    TEST TRIMLEN LENGTH RETURNED-DONE
                    12345678901234567890123456789012345678901234567890
```

Example 2: This example produces a string with the character < written following the last nonblank character of STRING:

```
      WRITE(6,901)STRING(1:TRIMLEN(STRING))
  901 FORMAT(' The string is >',A,'<')
```

Example 3: In this example, although NEW may have trailing blanks, the character < is written after the last nonblank character in STRING:

```
NEW = STRING(1:TRIMLEN(STRING)) // '< The end'
```

NAME

TSECND – Returns elapsed CPU time for a calling task during a multitasked program

SYNOPSIS

second=TSECND()

CALL TSECND(*second*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TSECND returns the elapsed CPU time of a calling task since the start of that task.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

second A real variable. *second* is the elapsed CPU time in seconds.

result A real variable, the same as *second* (this is optional for function calls).

The initial call to TSECND may take longer than subsequent calls due to certain initializations performed by the routine. If the cost of calling TSECND is important, ignore the initial call when computing TSECND's time.

TSECND only returns CPU time for the current task. It is your responsibility to determine which task is running at the time the call is made. This call may not be useful in an Autotasking environment, because the running user program does not know which task is actually executing the parallel regions of code.

For CRAY C90 series, CPU times returned by TSECND include wait-semaphore time. For all other systems, CPU times returned by TSECND do not include wait-semaphore time. Contrast this with the SECOND(3F) call.

EXAMPLES

The following example calculates how much of the total execution time for a multitasked program is accumulated by the calling process.

```
BEFORE = SECOND( )
TBEFORE = TSECND( )
CALL DOWORK( )                ! The subroutine DOWORK or
AFTER = SECOND( )             ! something it calls may be
TAFTER = TSECND( )            ! multitasked.
CPU = (AFTER - BEFORE)
TCPU = (TAFTER - TBEFORE)
MYPORTION = TCPU/CPU
```

SEE ALSO

RTC(3I), SECOND(3F), SECONDR(3F)

NAME

UNPACK – Expands stored data

SYNOPSIS

CALL UNPACK(*p, nbits, u, count*[*, sef*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

UNPACK takes 64-bit words and extracts partial word items and places them right-justified into 64-bit words. The items can be optionally sign-extended.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

p Vector of full 64-bit words to be expanded.

nbits The number of rightmost bits of data in each partial word; must be 1, 2, 4, 8, 16, or 32.

u On exit, vector of unpacked data.

count Number of resulting partial words.

sef Optional sign extension flag. If .TRUE., items are sign-extended. If omitted or .FALSE., items are not sign-extended.

SEE ALSO

PACK(3F)

NAME

AUXSTAT – Gives the number of reads and writes to SDS that have occurred while accessing auxiliary array variables

SYNOPSIS

```
CALL AUXSTAT()
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

AUXSTAT prints to stderr the number of reads and writes to secondary data segments (SDS) that have occurred while accessing auxiliary array variables. See *CF77 Commands and Directives*, publication SR–3771.

NAME

BITVEC – Generates a bit mask corresponding to integer or real arrays according to a specified condition

SYNOPSIS

```
INTEGER  BITVEC
INTEGER  np,  type,  nbits,  bits(*),  npop,  incv1,
incv2
INTEGER  v1(*),  v2(*)
  or
REAL  v1(*),  v2(*)
INTEGER  s
  or
REAL  s
np  =  BITVEC(type,  nbits,  bits,  npop,  v1,  incv1,  s)
   or
np  =  BITVEC(type,  nbits,  bits,  npop,  v1,  incv1,  v2,  incv2)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

BITVEC efficiently creates a bit mask that results from a comparison of elements of one vector with those of another vector or a scalar.

The resulting bit mask is padded with zeros to a word boundary.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following arguments are passed to BITVEC:

type Integer value, variable, or array element containing the zero-padded Hollerith-type code that defines the type of vector comparison that will be used to generate the bit mask.

If *type* begins with V, a vector-to-vector compare is implied. If *type* begins with S, a vector-to-scalar compare is implied.

The following values are valid for *type*:

type	**Value**
'VFEQ'L	*v1*.EQ.*v2* FLOATING-POINT
'VFNE'L	*v1*.NE.*v2* FLOATING-POINT
'VFGT'L	*v1*.GT.*v2* FLOATING-POINT
'VFGE'L	*v1*.GE.*v2* FLOATING-POINT
'VFLT'L	*v1*.LT.*v2* FLOATING-POINT
'VFLE'L	*v1*.LE.*v2* FLOATING-POINT
'VIEQ'L	*v1*.EQ.*v2* INTEGER
'VINE'L	*v1*.NE.*v2* INTEGER
'VIGT'L	*v1*.GT.*v2* INTEGER
'VIGE'L	*v1*.GE.*v2* INTEGER
'VILT'L	*v1*.LT.*v2* INTEGER
'VILE'L	*v1*.LE.*v2* INTEGER
'SFEQ'L	*v1*.EQ.*s* FLOATING-POINT
'SFNE'L	*v1*.NE.*s* FLOATING-POINT
'SFGT'L	*v1*.GT.*s* FLOATING-POINT
'SFGE'L	*v1*.GE.*s* FLOATING-POINT
'SFLT'L	*v1*.LT.*s* FLOATING-POINT
'SFLE'L	*v1*.LE.*s* FLOATING-POINT
'SIEQ'L	*v1*.EQ.*s* INTEGER
'SINE'L	*v1*.NE.*s* INTEGER
'SIGT'L	*v1*.GT.*s* INTEGER
'SIGE'L	*v1*.GE.*s* INTEGER
'SILT'L	*v1*.LT.*s* INTEGER
'SILE'L	*v1*.LE.*s* INTEGER

nbits Vector length. This defines the size of the input vectors in words and the output bit mask in bits.

bits Output bit mask. Must be word-aligned and be of size (*nbits*+63)/64 words.

npop Output number of bits set in bit mask. This parameter is assigned –1 if `BITVEC` was not called with correct parameters.

v1 Real or integer array that contains the first vector.

inv1 Memory stride in words for array *v1*. This stride may be negative.

v2 REAL or INTEGER array that contains the second vector if doing a vector-to-vector compare.

inv1 Memory stride for array *v2* in words. This stride may also be negative, and is used only if a vector-to-vector compare is requested.

s REAL or INTEGER constant, variable, or array element containing the second vector if doing a vector-to-scalar compare.

RETURN VALUES

Equivalent to *npop*. The number of bits set in the bit mask, or –1 if `BITVEC` was not called with correct parameters.

NAME

BITVECM – Generates a bit mask corresponding to masked integer arrays according to a specified condition

SYNOPSIS

```
INTEGER  BITVECM
INTEGER np, type, nbits, bits(*), npop, incv1, incv2
INTEGER v1(*), v2(*)
  or
INTEGER s
np = BITVECM(type, nbits, bits, npop, v1, incv1, mask1, shift1, s [, incv2,
mask2, shift2])
  or
np = BITVECM(type, nbits, bits, npop, v1, incv1, mask1, shift1, v2, incv2,
mask2, shift2)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

BITVECM efficiently creates a bit mask that results from a comparison of elements of one vector with those of another masked and shifted vector or a scalar.

The resulting bit mask is padded with zeros to a word boundary.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following arguments are passed to BITVECM:

type Integer value, variable, or array element containing the zero-padded Hollerith type code that defines the type of vector comparison that will be used to generate the bit mask.

If *type* begins with V, a vector-to-vector compare is implied. If *type* begins with S, a vector-to-scalar compare is implied.

The following values are valid for *type*:

type	**Value**
´VIEQ´L	*v1*.EQ.*v2* INTEGER
´VINE´L	*v1*.NE.*v2* INTEGER
´VIGT´L	*v1*.GT.*v2* INTEGER
´VIGE´L	*v1*.GE.*v2* INTEGER
´VILT´L	*v1*.LT.*v2* INTEGER
´VILE´L	*v1*.LE.*v2* INTEGER
´SIEQ´L	*v1*.EQ.*s* INTEGER
´SINE´L	*v1*.NE.*s* INTEGER
´SIGT´L	*v1*.GT.*s* INTEGER
´SIGE´L	*v1*.GE.*s* INTEGER
´SILT´L	*v1*.LT.*s* INTEGER
´SILE´L	*v1*.LE.*s* INTEGER

nbits Vector length. This defines the size of the input vectors in words and the output bit mask in bits.

bits Output bit mask. Must be word-aligned and be of size (*nbits*+63)/64 words.

npop Output number of bits set in bit mask. This argument is assigned a value of -1 if BITVECM was not called with correct parameters.

v1 INTEGER array that contains the first vector.

incv1 Memory stride in words for array *v1*. This stride can be negative.

mask1 Mask used to perform a bitwise-OR on the elements of *v1* after the shift.

shift1 Number of bits to right shift the elements of *v1* before masking.

v2 INTEGER array that contains the second vector if doing a vector-to-vector compare.

s INTEGER constant, variable, or array element containing the second vector if doing a vector-to-scalar compare.

incv2 Memory stride for array *v2* in words. This stride can also be negative, and is used only if a vector-to-vector compare is requested.

mask2 Mask used to perform a bitwise-OR on the elements of *v2* or *s* after the shift.

shift2 Number of bits to right shift the elements of *v2* or *s* before masking.

RETURN VALUES

Equivalent to *npop*. The number of bits set in the bit mask, or -1 if `BITVECM` was not called with correct parameters.

NAME

PUTBYT, IGTBYT – Replaces a byte in a variable or an array

SYNOPSIS

value=PUTBYT(*string, position, value*)

byte=IGTBYT(*string, position*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

PUTBYT replaces a specified byte in a variable or an array with a specified value. IGTBYT extracts a specified byte from a variable or an array.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

string The address of a variable or array. The variable or array may be of any type, except character.

position The number of the byte to be replaced or extracted. This argument must be an integer ≥ 1. If *position* is ≤ 0, no change is made to the destination string; *value* returned is -1. For IGTBYT, if *position* is ≥ 0, *value* is an integer between 0 and 255.

value The new value to be stored into the byte. This argument must be an integer with a value between 0 and 255.

If PUTBYT is called as an integer function (having been properly declared in the user program), the value of the function is the value of the byte stored.

The high-order 8 bits of the first word of the variable or array are called byte 1.

The value of the byte returned by IGTBYT is an integer value between 0 and 255.

NAME

DTTS – Converts ASCII date and time to time stamp

SYNOPSIS

```
INTEGER dtts
ts=DTTS(date, time, ts)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

DTTS converts ASCII date and time to time stamp.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ts Time stamp corresponding to *date* and *time* (type integer). On return, if *ts*=0, an incorrect parameter was passed to DTTS. The routine returns a positive value for dates and times in the proper format and between 01/01/73 00:00:01 and 12/31/99 23:59:59, inclusive.

date On entry, ASCII date in *mm/dd/yy* format. This may not be a Fortran character variable.

time On entry, ASCII time in *hh:mm:ss* format. This may not be a Fortran character variable.

NAME

ICPUSED – Gets task CPU time in real-time clock (RTC) ticks

SYNOPSIS

```
INTEGER itotalcp
itotalcp = ICPUSED( )
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ICPUSED gets the user CPU time used by the calling task in real-time clock ticks.

The accuracy of ICPUSED is not affected by system interrupts.

This function is equivalent to the TSECND(3F) function except it returns the time in RTC ticks rather than seconds; it returns the elapsed CPU time of the calling task.

For CRAY T90 and CRAY C90 series, CPU times returned by ICPUSED include wait-semaphore time. For all other systems, CPU times returned by ICPUSED do not include wait-semaphore time.

SEE ALSO

cpused(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

MTTS – Converts machine time (real-time clock value) to time stamp

SYNOPSIS

```
INTEGER MTTS, ts, irtc, cptype, cpcycl
ts=MTTS(irtc [, cptype, cpcycl])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

MTTS converts machine time (real-time clock value) to a time stamp.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ts 64-bit integer result containing a time stamp in nanoseconds.

irtc 64-bit integer real-time clock value to be converted to a time stamp in nanoseconds.

cptype CPU type (not implemented under the UNICOS operating system). The CPU type is the host machine.

cpcycl Integer specifying the CPU cycle time in picoseconds.

The integer function MTTS converts a real-time clock value *irtc* to a time stamp *ts* using *cpcycl*. If *cpcycl* is not present, MTTS uses the clock cycle time for the system on which the executable is running.

The input *ts* and the result *irtc* may be larger than 48 bits.

SEE ALSO

dtts(3F), tsdt(3F), tsmt(3F), unitts(3F)

NAME

MVC – Moves characters from one memory area to another

SYNOPSIS

CALL MVC(*s1, j1, s2, j2, k*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

MVC moves characters from one memory area to another.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

s1 Word address of the source string.

j1 Byte offset from the source string word address of the first byte of the source string (the high-order byte of the first word of the source string is byte 1).

s2 Word address of the destination string.

j2 Byte offset from the destination string word address of the first byte of the destination string (the high-order byte of the first word of the destination string is byte 1).

k Number of bytes to be moved.

Source and destination strings can occur on any byte boundary. The move is performed 1 character at a time, from left to right, so MVC is optimal only for certain types of overlapping moves.

EXAMPLES

To copy the first byte of an array throughout the array, invoke the routine as follows, where K is the length of the array in bytes:

```
CALL MVC(ARRAY,1,ARRAY,2,K-1)
```

NAME

P32, U32 – Packs or unpacks 32-bit words into or from Cray 64-bit words

SYNOPSIS

```
CALL P32(src, dest, num)
CALL U32(src, dest, num)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

P32 packs 32-bit words into Cray 64-bit words. U32 unpacks 32-bit words from Cray 64-bit words. The PACK(3F) and UNPACK(3F) routines provide similar functionality.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src For P32, a variable or array of any noncharacter type and of any length containing 32-bit words, left-justified in a Cray 64-bit word. For U32, a variable or array of any type or length containing 32-bit words as a continuous stream of data. Unpacking always starts with the leftmost bit of *src*.

dest For P32, a destination array of any type to contain the packed 32-bit words as a continuous stream of data. For U32, a destination array of any type to contain the unpacked 32-bit words, left-justified and sign-filled in a Cray 64-bit word.

num Number of 32-bit words to pack or unpack. The routine reads this many elements of *src* or generates this many elements of *dest.* Specify an integer variable, expression, or constant.

SEE ALSO

PACK(3F), UNPACK(3F)

NAME

P6460, U6064 – Packs or unpacks 60-bit words into or from Cray 64-bit words

SYNOPSIS

```
CALL P6460(src, dest, isb, num [,stride])
CALL U6064(src, isb, dest, num [,stride])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

P6460 packs 60-bit words into Cray 64-bit words. U6064 unpacks 60-bit words from Cray 64-bit words. Arguments must be addressed in the same order in which they appear in the SYNOPSIS section.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following arguments are used with this routine:

src Variable or array of any noncharacter type and of any length containing 60-bit words, left-justified in a Cray 64-bit word. For U6064, words are accepted as a continuous stream of data.

dest For P6460, a destination array of any type to contain the packed 60-bit words as a continuous stream of data. For U6064, a destination array of any type to contain the unpacked 60-bit words, left-justified and sign-filled in a Cray 64-bit word.

isb Bit location that is the leftmost storage location for the 60-bit words. Bit position is counted from the left to right, with the leftmost bit 0. Specify an integer variable, expression, or constant.

num Number of 60-bit words to pack or unpack. The routine reads this many elements of *src* or generates this many elements of *dest.* Specify an integer variable, expression, or constant.

stride Optional stride parameter for the *dest* array (U6064) or the *src* array (P6460). If not specified, a *stride* of 1 is used.

NAME

SECOND – Returns elapsed CPU time

SYNOPSIS

second=SECOND()

Cray PVP systems:

CALL SECOND(*second*)

IMPLEMENTATION

Cray PVP systems

SPARC systems

DESCRIPTION

SECOND returns the elapsed user CPU time (a real number in seconds) since the start of a program, including time accumulated by all processes in a multitasking program.

SECOND returns execution time only for the current program. For example, a script runs a 50-second program 10 times. A SECOND call at the end of the 10th run (or 1st or 3rd or 7th) returns 50 seconds.

On Cray PVP systems, SECOND is not appropriate for timing small timed regions. If the timed region is fewer than 4000 clock periods (CPs) in duration, the variation in the accuracy of SECOND can be 10% or more on one processor. On multiple processors, the variation can be on the order of 10,000 to 400,000 CPs.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

second Result; CPU time (in type real seconds) accumulated by all processes in a program.

NOTES

On Cray PVP systems, the initial call to SECOND may take longer than subsequent calls due to certain initializations performed by the routine. If the cost of calling SECOND is important, ignore the initial call when computing SECOND's time. The calculation of OVERHEAD in the second example, following, serves this purpose.

On Cray PVP systems, the CPU times gathered by SECOND include wait-semaphore time.

WARNINGS

If you are trying to time your entire program, and you are using Autotasking, the CPU time returned by SECOND may not reflect all of the CPU time used by your program. This is due to the fact that slave CPUs may be still active when your program terminates (for instance, with a STOP or END statement). Depending on the number of active CPUs on your machine, the extra time needed to terminate the slaves may be substantial compared with your program time. Thus, if you use SECOND to time your program, and you are also using time(1) or ja(1), the time reported by these other utilities could be much higher than that reported by SECOND.

Because of the large variation on multiple processors, SECOND should be used cautiously in multitasked routines.

On SPARC systems, the presence of multitasking does not affect the accuracy of SECOND.

EXAMPLES

Example 1: This example calculates the CPU time used in DOWORK:

```
BEFORE = SECOND( )
CALL DOWORK( )
AFTER = SECOND( )
CPUTIME = AFTER - BEFORE
```

Example 2: If the CPU time is small enough that the overhead for calling SECOND may be significant, the following example is more accurate:

```
T0 = SECOND( )
OVERHEAD = SECOND( ) - T0
BEFORE = SECOND( )
CALL DOWORK( )
AFTER = SECOND( )
CPUTIME = (AFTER - BEFORE) - OVERHEAD
```

SEE ALSO

RTC(3I), SECONDR(3F), TIMEF(3F), TSECND(3F)

ja(1), time(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

TREMAIN – Returns the CPU time (in seconds of type real) remaining for the program

SYNOPSIS

CALL TREMAIN(*result*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

This routine returns the CPU time remaining for the program.

result The calculated CPU time remaining. Type real.

NOTES

The time remaining is the lesser of the CPU time remaining for the current program, and the CPU time remaining for the current process.

TREMAIN returns 0 after the time limit has been exceeded and returns a large number if no CPU time limits are in effect.

NAME

TSDT – Converts time stamps to ASCII date and time strings

SYNOPSIS

CALL TSDT(*ts*, *date*, *hhmmss*, *ssss*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

TSDT converts time stamps to ASCII date and time.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ts Time stamp on entry (type integer). This may not be Fortran character variable.

date Word to receive ASCII date in *mm/dd/yy* format. This may not be Fortran character variable.

hhmmss Word to receive ASCII time in *hh:mm:ss* format. This may not be Fortran character variable.

ssss Word to receive ASCII fractional seconds in *.ssssnnn* format

NAME

TSMT – Converts time stamp to machine time (real-time clock value)

SYNOPSIS

```
INTEGER TSMT irtc, ts, cptype, cpcycl
irtc=TSMT(ts [, cptype, cpcycl])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

TSMT converts a time stamp to machine time (real-time clock value). The following are valid arguments for this routine.

irtc 64-bit integer result of the conversion of the specified time stamp to a real-time clock value.

ts 64-bit integer specifying a time stamp in nanoseconds to be converted to a real-time clock value.

cptype CPU type (not implemented under the UNICOS operating system). The CPU type is the host machine.

cpcycl Integer specifying CPU cycle time in picoseconds.

The integer function TSMT converts time stamp *ts* specified in nanoseconds to a number of clock ticks using *cpcycl*. If *cpcycl* is not present, TMST uses the clock cycle time for the system on which the executable is running.

The input *ts* and the result *irtc* may be larger than 48 bits.

SEE ALSO

DTTS(3F), MTTS(3F), TSDT(3F), UNITTS(3F)

NAME

UNITTS – Returns time-stamp units in specified standard-time units

SYNOPSIS

```
INTEGER  UNITTS
ts=UNITTS(periods, units)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

UNITTS returns time-stamp units in specified standard-time units.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ts Number of time-stamp units in *periods* and *units* (type integer).

periods Number of time-stamp units to be returned in standard-time units (that is, number of seconds, minutes, and so on); type integer.

units Specification for the units in which *periods* is expressed. The following values are accepted: 'DAYS', 'HOURS', 'MINUTES', 'SECONDS', 'MSEC' (milliseconds), 'USEC' (microseconds), 'USEC100' (100s of microseconds). UNITTS may be a character variable or type integer. If it is type integer, the values must be left-justified, blank-filled Hollerith.

EXAMPLES

```
ts=UNITTS(2,'DAYS')
```

NAME

`cinter` – Introduction to interfaces to C library routines

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This section contains descriptions of interfaces to C library routines.

A number of Fortran-callable interfaces to C library routines are available under the UNICOS operating system. These routines give a Fortran programmer access to an extensive number of routines and system calls found in the C library. The interfaces are simple routines that resolve calling sequence differences and provide uppercase entry-point names. Argument lists and return values should match those of the corresponding C routine, except where noted otherwise. Calling sequences for many of these Fortran interfaces to C routines are documented on the corresponding C library man page. Except where otherwise noted, data types should be handled as follows:

- C character data may be defined as a Fortran character data type for some of the C routines. See each man page for a description of the routines that allow this. For the routines that do not accept Fortran character data types as an argument, C character data should be defined as Fortran integers and terminated by a null (0) byte. On Cray PVP systems, Hollerith data handles this for 1–7 characters in length.
- C pointers should be handled by Fortran integers.
- Other C data types are compatible with their Fortran counterparts.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is `KIND=8` for integer, real, complex and logical arguments.

Interface routines should be coded as Fortran functions.

Example:

```
      CHARACTER*8 PATH, IDA(10)
      INTEGER FOPEN, FWRITE
      PATH='filenm  '
C     FOPEN returns an integer
      ISTREAM = FOPEN ( PATH, 'w+' )
      IF ( ISTREAM .EQ. 0 ) THEN
         PRINT *,'FOPEN failed'
         CALL ABORT
      ENDIF
      DO 10 I=1,10
         IDA(I)='testcase'
10    CONTINUE
      N=8
      J=FWRITE ( IDA(1), N, 10, ISTREAM )
         .
         .
      END
```

NOTES

The following set of interface routines is provided in the standard UNICOS libraries. See the appropriate Cray Research manuals or the appropriate online man page for specific usage information.

Purpose	Name	Heading
Closes or flushes a stream	`fclose` `fflush`	`fclose` `fflush`
Gets integer file descriptor associated with stream	`fileno`	`ferror`

Opens a stream	`fopen`
	`fdopen`
	`freopen`
		`fopen`

Gets a string from a stream	`fgets`	`gets`
Puts a string on a stream	`fputs`	`puts`
Performs binary I/O	`fread` `fwrite`	`fread`

Purpose	Name	Heading
Repositions a file pointer in a stream	`fseek` `ftell`	`fseek`
Returns the current working directory	`getcwd`	`getcwd`
Returns value for environment name	`getenv`†	`getenv`
Gets option letter from argument vector	`getopt`†	`getopt`
Makes a unique file name	`mktemp`	`mktemp`
Changes or adds value to the environment	`putenv`	`putenv`
Suspends execution for a specified interval	`sleep`	`sleep`
Creates a name for a temporary file	`tempnam`	`tempnam`
Creates a temporary file	`tmpfile`	`tmpfile`
Prints a traceback	`tracebk`	`tracebk`

† Not on Cray MPP systems or CRAY T90 series.

For the Fortran syntax for the above routines, see the appropriate page in the *UNICOS System Libraries Reference Manual*, publication SR–2080. For more information on Fortran I/O routines, see the *Application Programmer's I/O Guide*, publication SG–2168.

Many routines which are not supported on Cray MPP systems or CRAY T90 series are supported as POSIX 1003.9 routines. See the `INTRO_SPECIALIBS`(3F) man page for details.

Purpose	Name	Heading
Determines accessibility of a file	`access`†	`access`
Sets a process alarm clock	`alarm`	`alarm`
Changes working directory	`chdir`†	`chdir`
Changes mode of file	`chmod`†	`chmod`
Changes owner and group of a file	`chown`†	`chown`
Changes root directory	`chroot`†	`chroot`
Closes a file descriptor	`close`	`close`

Purpose	Name	Heading
Selects which processors may run the process	cpselect†††	cpselect
Creates a new file or rewrites an existing one	creat†	creat
Terminates a program and specifies status	exit	exit
Terminates all processes, not just the calling process	newexit†	exit
Controls open files	fcntl	fcntl
Creates a new process	fork†	fork
Gets effective group ID	getegid	getuid
Gets effective user ID	geteuid	getuid
Gets group ID	getgid	getgid
Gets process-group ID	getpgrp	getpid
Gets process ID	getpid	getpid
Gets parent process ID	getppid	getpid
Returns process ID of multitasking group	newgetpid†††	getpid
Returns process ID of the multitasking group that created the caller's multitasking group.	newgetppid†††	getpid
Gets user ID	getuid	getuid
Allocates storage for a file	ialloc	ialloc
Controls device	ioctl	ioctl
Sends a signal to a process or a group of processes	kill	kill
Sets resource limits	limit	limit
Creates a link to a file	link†	link
Moves read/write file pointer	lseek	lseek
Makes a directory, or a special or regular file	mknod†	mknod
Changes priority of process	nice	nice
Changes priority of process	nicem	nice

Purpose	Name	Heading
Opens a file for reading or writing	open†	open
Suspends process until signal	pause	pause
Creates an interprocess channel	pipe	pipe
Locks process in memory	plock†††	plock
Traces processes	ptrace†††	ptrace
Performs asynchronous read from a file	reada	reada
Controls execution of processes	resume	resume
Changes data segment space allocation	sbreak sbrk brk	brk
Sets group ID	setgid†††	setuid
Sets process-group ID	setpgrp	setpgrp
Sets user ID	setuid†††	setuid
Provides signal control	sigctl	sigctl
Fortran interface to signal control	fsigctl	
Pascal interface to signal control	psigctl†	
Specifies what to do upon receipt of a signal	signal	signal
Fortran interface to signal	fsignal	
Pascal interface to signal	psignal	
Manages the allocation and deallocation of the secondary data segment (SDS)	sdsalloc†† sdsrealc†† sdsfree††	sdsalloc
Changes size of secondary data segment (Fortran users are encouraged to use the new SDSALLOC routines instead of ssbreak. See the sdsalloc(3F) man page.)		ssbreak††
Reads, writes to secondary data segment	ssread†† sswrite	ssread

Purpose	Name	Heading
Gets file status	stat†	stat
Controls execution of processes	suspend	suspend
Flushes system buffers out of main memory	sync	sync
Gets time	time	time
Gets process and child process times	times	times
Truncates a file	trunc	trunc
Gets and sets user limits	ulimit	ulimit
Sets and gets file creation mark	umask	umask
Gets name of current operating system	uname†	uname
Removes directory entry	unlink†	unlink
Gets file system statistics	ustat	ustat
Sets file access and modification times	utime†	utime
Awaits completion of process	wait†	wait
Performs asynchronous write on a file	writea	writea

† Not available on Cray MPP systems or CRAY T90 series.
†† See sdsalloc(3F) for restrictions when using these routines on Cray MPP systems.
††† Not available on Cray MPP systems.

The argument lists of the uname(3) and time(3) routines differ from those of the corresponding C routines. No arguments can be used with the Fortran call to time. See the man page in this manual for the correct syntax when calling uname from Fortran.

The third argument of Fortran routines ssread(2) and sswrite(3) specifies the number of words to be read or written. This is different from the corresponding system call. See the WARNING section of the SDSALLOC(3F) routine in this manual for cautions on using ssbreak(2) in Fortran programs.

For more information on Fortran I/O routines, see the *Application Programmer's I/O Guide*, publication SG–2168.

General Usage Example

When a system call or library function has a pointer to a data structure as an argument, the corresponding Fortran callable routine requires a pointer to a data entity at least as large as the structure. Any exception to this requirement is documented in the specific man page.

Accessing the fields of a structure from Fortran may not be possible when the structure contains multiple data types or when the structure uses a data type not recognized by Fortran.

For a few library routines and system calls, alternate Fortran callable entry points provide a simpler method of accessing structures. These entry points being with the prefix `pxf`. See `pxfintget`(3F) for details.

An example of a call to the `UTIME()` system call from a Fortran program follows. The second argument to `UTIME` is a pointer to a structure of type `utimbuf`. This structure is defined as follows:

```
struct utimbuf {
          time_t actime;
          time_t modtime;
}
```

In the following example, the Fortran programmer uses an integer array to hold the structure. The first word of the array contains the `actime` field; the second word contains the `modtime` field.

```
      PROGRAM UTIM
      INTEGER TIME, UTIME
      INTEGER UT(2)
      I = TIME()
      UT(1) = I
      UT(2) = I
C     Update the access and modification times of file 'myfile'
C     to the current time
      I = UTIME('myfile', UT)
      IF (I.EQ.-1) THEN
          PRINT *, 'UTIME FAILED'
      ENDIF
      END
```