TruCluster Production Server Software

MEMORY CHANNEL Application Programming Interfaces

Part Number: AA-QTN4C-TE

January 1998

Product Version:	TruCluster Production Server Software Version 1.5 and TruCluster MEMORY CHANNEL Software Version 1.5
Operating System and Version:	DIGITAL UNIX Version 4.0D

This manual describes how to install the MEMORY CHANNEL software, and how to develop applications that are based on the MEMORY CHANNEL Application Programming Interface (API) library.

Digital Equipment Corporation
Maynard, Massachusetts

Contents

About This Manual

1 Software Installation

1.1 Installing the MEMORY CHANNEL Software 1–2
1.1.1 Obtain IP Names and Addresses 1–2
1.1.2 Halt System and Set Console Variables 1–3
1.1.3 Boot to Single-User Mode and Deinstall Software 1–4
1.1.4 Install the DIGITAL UNIX Operating System 1–4
1.1.5 Register the MEMORY CHANNEL Software License 1–5
1.1.6 Load the Kit 1–6
1.1.7 Specify a Network Interface IP Name and Address 1–7
1.1.8 Select a Kernel Configuration File 1–8
1.1.9 Build and Install a New Kernel 1–8
1.1.10 Reboot the System 1–9
1.1.11 Installation Example 1–9
1.1.12 Verify the Installation with the clu_ivp Utility 1–11
1.2 Initializing the MEMORY CHANNEL API Library 1–12
1.3 The MEMORY CHANNEL Multirail Model 1–13
1.3.1 Single-Rail Style 1–13
1.3.2 Failover Pair Style 1–14
1.3.3 Configuring the MEMORY CHANNEL Multirail Model .. 1–15
1.4 Tuning Your MEMORY CHANNEL Configuration 1–16
1.4.1 Extending MEMORY CHANNEL Address Space 1–17
1.4.2 Increasing Wired Memory 1–17
1.4.3 Increasing Virtual Memory Map Entries 1–18
1.5 Troubleshooting 1–18
1.5.1 IMC_NOTINIT Return Code 1–18
1.5.2 MEMORY CHANNEL API Library Initialization Failure 1–19
1.5.3 Fatal MEMORY CHANNEL Errors 1–20
1.5.3.1 Logical Rail Failure 1–20
1.5.3.2 Logical Rail Initialization Failure 1–21
1.5.3.3 MEMORY CHANNEL Cables Crossed 1–21
1.5.4 IMC_MCFULL Return Code 1–22
1.5.5 IMC_RXFULL Return Code 1–22

1.5.6 IMC_WIRED_LIMIT Return Code 1–22
1.5.7 IMC_MAPENTRIES Return Code 1–22
1.5.8 IMC_NOMEM Return Code 1–23
1.5.9 IMC_NORESOURCES Return Code 1–23

2 Application Notes

2.1 Initializing the MEMORY CHANNEL API Library for a User Program 2–1
2.2 Accessing MEMORY CHANNEL Address Space 2–2
2.2.1 Attaching to MEMORY CHANNEL Address Space 2–3
2.2.1.1 Broadcast Attach 2–4
2.2.1.2 Point-to-Point Attach 2–5
2.2.1.3 Loopback Attach 2–6
2.2.2 Initial Coherency 2–7
2.2.3 Reading and Writing MEMORY CHANNEL Regions 2–8
2.2.4 Address Space Example 2–8
2.2.5 Latency Related Coherency 2–11
2.2.6 Error Management 2–14
2.3 Clusterwide Locks 2–19
2.4 Cluster Signals 2–22
2.5 Cluster Information 2–22
2.5.1 Using MEMORY CHANNEL API Functions to Access MEMORY CHANNEL API Cluster Information 2–22
2.5.2 Accessing MEMORY CHANNEL Status Information from the Command Line 2–24
2.6 Comparison of Shared Memory and Message Passing Models 2–24

3 MEMORY CHANNEL API Library Interface

3.1 Header Files 3–1
3.2 Library 3–1
3.3 Compiling Applications that Use the MEMORY CHANNEL API Library 3–2
3.4 Overview of MEMORY CHANNEL API Library Commands and Functions 3–2
imc(3) 3–3
3.5 Command Descriptions 3–7
imc_init(1) 3–8
imcs(1) 3–11
3.6 Function Descriptions 3–15
imc_api_init(3) 3–16
imc_asalloc(3) 3–18

imc_asattach(3) 3–23
imc_asattach_ptp(3) 3–29
imc_asdealloc(3) 3–33
imc_asdetach(3) 3–35
imc_bcopy(3) 3–37
imc_ckerrcnt(3) 3–42
imc_ckerrcnt_mr(3) 3–44
imc_getclusterinfo(3) 3–46
imc_kill(3) 3–50
imc_lkacquire(3) 3–52
imc_lkalloc(3) 3–55
imc_lkdealloc(3) 3–59
imc_lkrelease(3) 3–61
imc_perror(3) 3–63
imc_rderrcnt(3) 3–65
imc_rderrcnt_mr(3) 3–67
imc_wait_cluster_event(3) 3–69

A Frequently Asked Questions

A.1 IMC_NOMAPPER Return Code A–1
A.2 Efficient Data Copy A–1
A.3 MEMORY CHANNEL Bandwidth Availability A–2
A.4 MEMORY CHANNEL API Cluster Configuration Change A–2
A.5 Bus Error Message A–2
A.6 Deciding Which TruCluster Product To Use A–2
A.7 Finding Out More About MEMORY CHANNEL A–3

Index

Examples

1–1 MEMORY CHANNEL Software Installation 1–9
2–1 Accessing Regions of MEMORY CHANNEL Address Space .. 2–9
2–2 System V IPC and MEMORY CHANNEL Code Comparison . 2–12
2–3 Error Detection Using the imc_rderrcnt_mr Function 2–16
2–4 Error Detection Using the imc_ckerrcnt_mr Function 2–18
2–5 Locking MEMORY CHANNEL Regions 2–19
2–6 Requesting MEMORY CHANNEL API Cluster Information; Waiting for MEMORY CHANNEL API Cluster Events 2–23

Figures

1–1 Single-Rail MEMORY CHANNEL Configuration 1–14

1–2 Failover Pair MEMORY CHANNEL Configuration 1–15
2–1 Broadcast Address Space Mapping 2–4
2–2 Point-to-Point Address Space Mapping 2–5
2–3 Loopback Address Space Mapping 2–7

About This Manual

This manual describes the TruCluster™ MEMORY CHANNEL™ Application Programming Interface (API) and how to use it to develop programs for TruCluster systems based on the MEMORY CHANNEL interconnect technology.

Audience

This manual is for developers who want to directly access the features provided by the MEMORY CHANNEL API, and for system managers who want to install the product. The manual assumes that the reader is experienced with the following:

- UNIX® operating environment
- C programming language
- Shared memory and message-passing programming models

Organization

This manual has three chapters, an appendix, and an index. The manual is structured as follows:

Chapter 1	Describes the MEMORY CHANNEL software installation procedure, initialization, and configuration.
Chapter 2	Contains information to help you develop applications based on the MEMORY CHANNEL API library.
Chapter 3	Provides reference information about each MEMORY CHANNEL API command and function.
Appendix A	Contains answers to questions asked by programmers who use the MEMORY CHANNEL API library.

Related Documents

Consult the following TruCluster Software Products manuals for assistance in configuring, installing, and administering the software:

- TruCluster Software Products *Release Notes*—Documents known restrictions and other important information about the TruCluster MEMORY CHANNEL Software product.
- TruCluster Software Products *Hardware Configuration*—Describes how to set up the processors to utilize the MEMORY CHANNEL hardware.

In addition, you should have available the following manuals from the DIGITAL UNIX documentation set:

- *Installation Guide*
- *Network Administration*
- *System Administration*

Location of Code Examples

This manual includes code examples that show how to use the MEMORY CHANNEL API library functions in programs. You will find these code files in the `/usr/examples/cluster/` directory. Each file contains compilation instructions.

Location of Online Documentation

Each book in the TruCluster Software documentation set is shipped as a set of Hypertext Markup Language (HTML) and graphics files in the `/TCR/doc/html` directory on the Associated Products Volume 2 CD-ROM. You can use the Netscape Navigator® World Wide Web browsing program to display these books.

To access the TruCluster Software documentation from the viewer, click on the Open icon in the Netscape main window and enter the following file location in the Open Location: text entry field: `file:/`*`mountpoint`*`/TCR/doc/html/BOOKSHELF.HTM`.

Reader's Comments

DIGITAL welcomes any comments and suggestions you have on this and other DIGITAL UNIX manuals. A Reader's Comment form is located on your system in the following location:

```
/usr/doc/readers_comment.txt
```

You can send your comments in the following ways:

- Internet electronic mail: `comments@ilo.dec.com`
- Fax: +353 91 754784 Attn: DIGITAL Information Design
- Mail:

  Digital Equipment Corporation
  DIGITAL Information Design
  Ballybrit Industrial Park
  Galway
  Ireland

  A Reader's Comment form is located in the back of each printed manual.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of DIGITAL UNIX that you are using.
- If known, the type of processor that is running the DIGITAL UNIX software.

The DIGITAL UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate DIGITAL technical support office. Information provided with the software media explains how to send problem reports to DIGITAL.

Conventions

The following typographical conventions are used in this manual:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
`#`	A number sign represents the superuser prompt.
`% cat`	Boldface type in interactive examples indicates typed user input.

file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|]	
{|}	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
cat(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the cat(1) command in Section 1 of the reference pages.
Return	In an example, a key name enclosed in a box indicates that you press that key.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

1

Software Installation

This chapter describes how to install the TruCluster MEMORY CHANNEL Software on the DIGITAL UNIX Version 4.0D operating system. It also describes how to initialize the MEMORY CHANNEL Application Programming Interface (API) library, and discusses MEMORY CHANNEL configuration.

Note

If you want to install and configure the MEMORY CHANNEL API in a TruCluster Production Server environment, you must use the TruCluster Software Products *Software Installation* manual.

The chapter discusses the following topics:

- Installing the MEMORY CHANNEL software (Section 1.1)
- Initializing the MEMORY CHANNEL API library (Section 1.2)
- The MEMORY CHANNEL multirail model (Section 1.3)
- Tuning your MEMORY CHANNEL configuration (Section 1.4)
- Troubleshooting (Section 1.5)

Note

Throughout this manual, a cluster is defined as a MEMORY CHANNEL API cluster, *not* a Production Server cluster. However, a Production Server cluster may be identical to the MEMORY CHANNEL API cluster.

Members of a MEMORY CHANNEL API cluster must be connected by a MEMORY CHANNEL interconnect, and must have executed the `imc_init` command. Production Server cluster membership criteria are not the same as MEMORY CHANNEL API cluster membership criteria. Production Server cluster membership is monitored by the connection manager. The connection manager is not associated with MEMORY CHANNEL API cluster membership.

1.1 Installing the MEMORY CHANNEL Software

This section and the following subsections describe how to prepare for the MEMORY CHANNEL software installation, the steps involved in the installation, and how to test the completed installation to make sure that it is working correctly.

Note

The procedures described in this section assume that each system's hardware and firmware are installed and configured as described in the TruCluster Software Products *Hardware Configuration* manual. Do not begin the software installation until the hardware and firmware are installed and configured.

Please note the following general installation restrictions and considerations:

- Do not install the product into a dataless environment.
- It is recommended that you have at least 64 MB of memory available on each member system.
- You must have superuser (root) privileges for the systems on which you will install the software.
- Back up all systems before beginning the installation process.
- The installation procedure automatically modifies the `/etc/sysconfigtab` file; however, in some cases this manual will tell you to modify, or add to, certain stanzas in the `/etc/sysconfigtab` file. DIGITAL recommends that you use `sysconfigdb`(8) to do this.

1.1.1 Obtain IP Names and Addresses

You must assign an Internet Protocol (IP) address and corresponding name to the MEMORY CHANNEL interface on each host.

The network is visible only to the hosts which are directly connected by the MEMORY CHANNEL interconnect; this means that you can use IP addresses of the form `10.0.0.x`, since this form of IP address is reserved for private networks. For example, for four hosts, you can assign the following IP addresses:

```
10.0.0.1
10.0.0.2
10.0.0.3
10.0.0.4
```

Note

Host number 42 (that is, IP address 10.0.0.42) is reserved by TruCluster software and must not be used. 10.0.0.64 is also a reserved IP address and must not be used.

See the DIGITAL UNIX *Network Administration* manual for detailed guidelines on allocating IP addresses.

You can assign any unique IP name to the MEMORY CHANNEL interface on each host; for example, you could use the IP name `mcclu` for a host named `clu`. Do not use an underscore (_) in an IP name.

The MEMORY CHANNEL software installation procedure updates the `/etc/hosts` and `/etc/rc.config` file to reflect the IP names and addresses you supply during installation.

The system's host name (the one displayed by the `hostname` program) does not change as a result of installing the MEMORY CHANNEL software.

1.1.2 Halt System and Set Console Variables

To halt the system and set the console variables, follow these steps:

1. Halt the system. For example, to halt the system from multiuser mode with no other users on the system, enter the following command:

   # **shutdown -h now**

2. If your system supports the `bus_probe_algorithm` console variable, set its value to `new`. This ensures that peripheral component interconnect (PCI) devices are consistently probed on all member systems. To check the setting, enter the following command at the console prompt:

   >>> **show bus_probe_algorithm**

   ```
   bus_probe_algorithm new
   ```

   If necessary, enter the following command to set the `bus_probe_algorithm` variable to `new`:

   >>> **set bus_probe_algorithm new**

3. In order to bring the system to a known state at each reboot, set the `boot_reset` console variable as follows:

   >>> **set boot_reset on**

1.1.3 Boot to Single-User Mode and Deinstall Software

If there are MEMORY CHANNEL subsets on the system, boot `/genvmunix` to single-user mode and deinstall these subsets by following these steps:

1. From the console prompt, boot `/genvmunix` to single-user mode; for example:

   >>> **`boot -fl s -fi /genvmunix`**

2. Enter the `bcheckrc` command, which makes the `root` file system writable and mounts local file systems:

   # **`bcheckrc`**

3. To make sure that the system's licenses are loaded and active, run the following LMF commands:

   # **`lmf reset`**
   # **`lmf list`**

4. Use the `setld -i` command to determine which MEMORY CHANNEL software subsets are installed.

5. Use the `setld -d` command to deinstall the subsets.

   To ensure that the subsets are deleted in an order that resolves any dependencies between subsets, delete all installed subsets with one `setld -d` command. The following example shows how to delete existing subsets:

   # **`setld -d TCRCONFnnn TCRMANnnn TCRMCAnnn TCRCOMMONnnn`**

   In the example, `nnn` represents the version number of the existing subsets that are to be deleted.

1.1.4 Install the DIGITAL UNIX Operating System

The first step in the MEMORY CHANNEL software installation is to install DIGITAL UNIX on each computer in the MEMORY CHANNEL API cluster. It is recommended that you install the same version of DIGITAL UNIX on all of the computers; this will create a more consistent clusterwide environment.

Note

You must install DIGITAL UNIX Version 4.0D or greater in order to run TruCluster MEMORY CHANNEL Software Version 1.5.

Before starting the installation procedures described in the DIGITAL UNIX *Installation Guide*, read the following list and incorporate these tasks into the installation:

- Before installing the operating system, turn on the power to all member systems, the MEMORY CHANNEL hub, and external disks.
- Load the following operating system subsets:

OSFBASE	Base System
OSFBIN	Standard Kernel Objects
OSFBINCOM	Kernel Header and Common Files
OSFHWBIN	Hardware Kernel Modules
OSFHWBINCOM	Hardware Kernel Header and Common Files
OSFCLINET	Basic Networking Services
OSFCMPLRS	Compiler Back End
OSFNFS	NFS® Utilities
OSFDCMT	Documentation Preparation Tools®

Important Note

If you install new hardware (for example, new MEMORY CHANNEL adapters) after you install or update the DIGITAL UNIX operating system, you must boot `/genvmunix` and build a customized kernel. Otherwise, the system's kernel configuration file will not contain these hardware options, and the kernel you build during MEMORY CHANNEL installation will not recognize the new hardware.

You must also rebuild the kernel if you change a MEMORY CHANNEL adapter from the PCI slot with which the original kernel was configured.

See the DIGITAL UNIX *System Administration* manual for more information on configuring kernels.

If you are performing an update installation of the DIGITAL UNIX Version 4.0D operating system, boot `/genvmunix` after installing the DIGITAL UNIX Version 4.0D operating system and before loading the MEMORY CHANNEL Version 1.5 software.

1.1.5 Register the MEMORY CHANNEL Software License

Before you install the MEMORY CHANNEL software, you must register its Product Authorization Key (PAK). The name of the PAK is **MCA-UA**.

Note

You must register the PAK before installing the MEMORY CHANNEL software; if the PAK is not registered, the installation procedure displays the following message:

```
There are no TruCluster Software licenses
installed. In order to install a TruCluster
product you must first install the appropriate LMF
PAK (TCR-UA or MCA-UA or ASE-OA).
```

You can register the PAK using the `lmfsetup` script or the `lmf register` command.

Note

The TCR-UA PAK is associated with TruCluster Production Server software; the ASE-OA PAK is associated with the TruCluster Available Server software. These PAKs should not be present. If you wish to install TruCluster Production Server software or TruCluster Available Server software, use the TruCluster Software Products *Software Installation* manual.

1.1.6 Load the Kit

To load the MEMORY CHANNEL software, follow these steps:

1. Log in as superuser.
2. Change the directory to root (`cd /`).
3. Mount the device or directory containing the MEMORY CHANNEL kit.
4. Enter the `setld -l` command and specify the directory where the kit is located. For example:

   `# setld -l /TCR150/kit`

   The installation procedure automatically starts and lists the available mandatory and optional subsets. You can choose one of the following subset installation options:

   - All mandatory subsets only
   - All mandatory and selected optional subsets
   - All mandatory and all optional subsets

   DIGITAL recommends that you choose the "All mandatory and all optional subsets" option.

After you select an option, the installation procedure checks that there is sufficient file system space. After this check completes, the installation procedure copies the subsets onto your system. (The following directories are the default locations for the majority of installed files: `/opt/TCR150/`, `/usr/opt/TCR150/`, and `/var/opt/TCR150/`. The `/usr/opt/TCR150/sbin/clu_install` script controls most of the installation process.)

Note

You cannot install individual product subsets. For example, the following command results in an error:

```
# setld -l /TCR150/kit/TCRCONF150
TCRCONF150 cannot be installed. Please do not install subsets
individually.
```

1.1.7 Specify a Network Interface IP Name and Address

The installation procedure prompts you for an Internet Protocol (IP) name and address to associate with the system's network interface. (See Section 1.1.1 for information about required IP names and addresses.)

In the following example, the MEMORY CHANNEL IP name is formed by adding the prefix `mc` with the current host name (`clu14`) to identify this IP name as an interface to the MEMORY CHANNEL subnet:

```
Configuring "TruCluster Configuration Software" (TCRCONF150)
Enter the IP NAME for the cluster interconnect: mcclu14
```

Note

If you make a mistake when specifying the IP name for the MEMORY CHANNEL adapter, press Return when prompted for the IP address. The installation procedure will prompt you for a new IP name.

The installation procedure reads the system's `/etc/hosts` file to determine whether an entry exists for the IP name. If an entry for the IP name exists, the installation procedure displays the entry and asks whether you want to replace the existing entry with the IP name and address you just specified. For example:

- If you want to use the existing `/etc/hosts` entry, answer **n**; the information you specified during installation is ignored.

- If you want to replace the existing /etc/hosts entry, answer **y**. The installation procedure then replaces the entry in the /etc/hosts file with the IP name and address you specified.

The installation procedure automatically configures the network interface.

1.1.8 Select a Kernel Configuration File

At this point in the installation process, the kernel configuration and build procedure begins. You are prompted for the name of a kernel configuration file. You can accept the default or enter the name of another configuration file. In the following example, the default configuration file, CLU14, is accepted:

```
The kernel will now be configured using "doconfig".
Enter the name of the kernel configuration file. [CLU14]: [Return]
```

After you specify the name of the kernel configuration file, the installation procedure asks whether you want to edit the file (after first saving the original configuration file with a .bck extension):

```
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
Saving /sys/conf/CLU14 as /sys/conf/CLU14.bck
Do you want to edit the configuration file? (y/n) [n]: [Return]
```

To edit the kernel configuration file, answer **y**. Otherwise, accept the default response (**n**). If you answer **y** and the EDITOR shell environment variable is defined, doconfig starts that editor; otherwise, it starts ed.

1.1.9 Build and Install a New Kernel

The doconfig program names the new kernel /sys/*filename*/vmunix, where *filename* is the name of the configuration file you specified when you configured the MEMORY CHANNEL API cluster kernel components (see Section 1.1.8).

If the kernel build is successful, the name of the new kernel file is displayed as follows:

```
        Working....Tue Nov 25 17:15:32 GMT 1997
        Working....Tue Nov 25 17:17:33 GMT 1997
        Working....Tue Nov 25 17:19:34 GMT 1997

The new kernel is /sys/CLU14/vmunix
```

When the kernel build is successful, the installation procedure displays a list of instructions; see Example 1–1 for details.

The installation procedure does not automatically move the new kernel to the root directory. You can rename the new kernel or save the existing kernel before manually moving the new kernel to the root directory.

Before moving the original kernel aside and copying the new one to the root directory, use the `df` command to check that there is enough disk space for both files.

Move the new kernel to the root directory. In the following example, the old kernel is saved as `vmunix.save` and the new kernel, `/sys/CLU14/vmunix`, is moved to the root directory:

```
# cp /vmunix /vmunix.save
# mv /sys/CLU14/vmunix /
```

After you verify the proper operation of the new kernel, you can remove the old kernel (called `vmunix.save` in this example). DIGITAL recommends that you keep a kernel that does not contain MEMORY CHANNEL API cluster support (for example, `/genvmunix`).

1.1.10 Reboot the System

To reboot the system, enter the following command:

```
# shutdown -r now
```

During the reboot, startup messages are displayed on the console.

To check the version of the installed software, enter the following command:

```
# sysconfig -q clubase cluster_version
```

1.1.11 Installation Example

Example 1–1 outlines a typical MEMORY CHANNEL software installation.

Example 1–1: MEMORY CHANNEL Software Installation

```
# cd /mnt/TCR150
# ls
TCRCOMMON150  TCRMCA150    TCRCONF150    TCRMAN150     instctrl
# setld -l .

*** Enter subset selections ***

The following subsets are mandatory and will be installed automatically
unless you choose to exit without installing any subsets:

      * TruCluster Common Components
      * TruCluster Configuration Software
      * TruCluster MEMORY CHANNEL(TM) Software

The subsets listed below are optional:

There may be more optional subsets than can be presented on a single
screen. If this is the case, you can choose subsets screen by screen
or all at once on the last screen. All of the choices you make will
```

Example 1–1: MEMORY CHANNEL Software Installation (cont.)

```
be collected for your confirmation before any subsets are installed.

 - TruCluster(TM) Software:
     1) TruCluster Reference Pages

Or you may choose one of the following options:

     2) ALL mandatory and all optional subsets
     3) MANDATORY subsets only
     4) CANCEL selections and redisplay menus
     5) EXIT without installing any subsets

Enter your choices or press RETURN to redisplay menus.
Choices (for example, 1 2 4-6): 2

You are installing the following mandatory subsets:

        TruCluster Common Components
        TruCluster Configuration Software
        TruCluster MEMORY CHANNEL(TM) Software

You are installing the following optional subsets:

 - TruCluster(TM) Software:
        TruCluster Reference Pages

Is this correct? (y/n): y

Checking file system space required to install selected subsets:

File system space checked OK.

4 subset(s) will be installed.

Loading 1 of 4 subset(s)....

TruCluster Common Components
   Copying from . (disk)
   Verifying

Loading 2 of 4 subset(s)....

TruCluster MEMORY CHANNEL(TM) Software
   Copying from . (disk)
   Verifying

Loading 3 of 4 subset(s)....

TruCluster Reference Pages
   Copying from . (disk)
   Verifying

Loading 4 of 4 subset(s)....

TruCluster Configuration Software
   Copying from . (disk)
   Verifying

4 of 4 subset(s) installed successfully.
```

Example 1–1: MEMORY CHANNEL Software Installation (cont.)

```
Configuring "TruCluster Common Components " (TCRCOMMON150)

Configuring "TruCluster MEMORY CHANNEL(TM) Software" (TCRMCA150)

Configuring "TruCluster Reference Pages " (TCRMAN150)

Configuring "TruCluster Configuration Software " (TCRCONF150)

Enter the IP name for the MEMORY CHANNEL adapter:mcclu14

Now you must enter an IP address corresponding to mcclu14.

Enter the IP address for mcclu14 ([Return] to restart): 10.0.0.1

You chose "mcclu14," IP 10.0.0.1 using interface mc0
Is this correct? [y]:  y

The kernel will now be configured using "doconfig".

Enter the name of the kernel configuration file. [CLU14]: CLU14

*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

Saving /sys/conf/CLU14 as /sys/conf/CLU14.bck

Do you want to edit the configuration file? (y/n) [n]: n


*** PERFORMING KERNEL BUILD ***
        Working....Tue Nov 25 17:15:32 GMT 1997
        Working....Tue Nov 25 17:17:33 GMT 1997
        Working....Tue Nov 25 17:19:34 GMT 1997

The new kernel is /sys/CLU14/vmunix

The kernel build was successful. Please perform the following actions:

        o Move the new kernel to /.
        o Before rebooting make sure that the MEMORY CHANNEL adapter IP
          addresses for all cluster members are recorded in each member's
          /etc/hosts file.
        o Reboot the system.

# mv /sys/CLU14/vmunix /
# reboot
```

1.1.12 Verify the Installation with the clu_ivp Utility

After the MEMORY CHANNEL software is installed or upgraded, use the cluster installation verification program, `clu_ivp`, to detect configuration errors.

By default, the `clu_ivp` utility displays error conditions only. When an error is detected, the `clu_ivp` utility suggests corrective action. In some cases, the error reported by the `clu_ivp` utility is the symptom of another

problem. Read all the error messages generated by the `clu_ivp` utility before attempting to correct problems. When the corrective action suggested by the `clu_ivp` utility does not solve the problem, examine the system's error log files and console output for additional clues.

For more informative output, use the `clu_ivp -v` (verbose) option. In addition to reporting error conditions, the utility displays confirmation of each verification check as it is performed.

1.2 Initializing the MEMORY CHANNEL API Library

To run applications based on the MEMORY CHANNEL API library, the library must be initialized on each host in the MEMORY CHANNEL API cluster. The `imc_init` command initializes the MEMORY CHANNEL API library and allows applications to use the API. Initialization of the MEMORY CHANNEL API library occurs either by automatic execution of the `imc_init` command at system boot time, or when the system administrator invokes the command from the command line after the system boots.

Initialization of the MEMORY CHANNEL API library at system boot time is controlled by the `IMC_AUTO_INIT` variable in the `/etc/rc.config` file. If the value of this variable is set to 1, the `imc_init` command is invoked at system boot time. When the MEMORY CHANNEL API library is initialized at boot time, the values of the `-a` *maxalloc* and `-r` *maxrecv* flags are set to the values specified by the `IMC_MAX_ALLOC` and `IMC_MAX_RECV` variables in the `/etc/rc.config` file. The default value for the *maxalloc* parameter and the *maxrecv* parameter is 10 MB.

If the `IMC_AUTO_INIT` variable is set to zero (0), the MEMORY CHANNEL API library is not initialized at system boot time. The system administrator must invoke the `imc_init` command to initialize the library. The parameter values in the `/etc/rc.config` file are not used when the `imc_init` command is manually invoked.

The `imc_init` command initializes the MEMORY CHANNEL API library the first time it is invoked, whether this happens at system boot time or after the system has booted. The value of the `-a` *maxalloc* flag must be the same on all hosts in the MEMORY CHANNEL API cluster. If different values are specified, the maximum value specified for any host determines the clusterwide value that applies to all hosts.

After the MEMORY CHANNEL API library has initialized on the current host, the system administrator can invoke the `imc_init` command again to reconfigure the values of the *maxalloc* and *maxrecv* resource limits, without forcing a reboot. The system administrator can increase or decrease either limit, but the new limits cannot be lower than the current usage of the resources. Reconfiguring the cluster from the command line

does not read or modify the values specified in the `/etc/rc.config` file. The system administrator can use the `rcmgr`(8) command to modify the parameters and have them take effect when the system reboots.

You must have root privileges to execute the `imc_init` command.

1.3 The MEMORY CHANNEL Multirail Model

The MEMORY CHANNEL multirail model supports the concept of physical rails and logical rails. A physical rail is defined as a MEMORY CHANNEL hub with its cables and MEMORY CHANNEL adapters and the MEMORY CHANNEL driver for the adapters on each node. A logical rail is made up of one or two physical rails.

A cluster can have one or more logical rails, up to a maximum of four. Logical rails can be configured in the following styles:

- Single-rail
- Failover pair

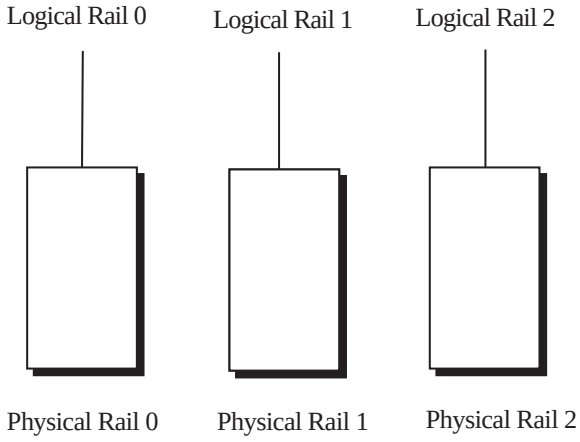
1.3.1 Single-Rail Style

If a cluster is configured in the single-rail style, there is a one-to-one relationship between physical rails and logical rails. This configuration has no failover properties; if the physical rail fails, the logical rail fails.

A benefit of the single-rail configuration is that applications can access the aggregate address space of all logical rails and utilize their aggregate bandwidth for maximum performance.

Figure 1–1 shows a single-rail MEMORY CHANNEL configuration with three logical rails, each of which is also a physical rail.

Figure 1–1: Single-Rail MEMORY CHANNEL Configuration



1.3.2 Failover Pair Style

If a cluster is configured in the failover pair style, a logical rail consists of two physical rails, with one physical rail active and the other inactive. If the active physical rail fails, a failover takes place and the inactive physical rail is used, allowing the logical rail to remain active after the failover. This failover is transparent to the user.

The failover pair style can only exist in a MEMORY CHANNEL configuration consisting of two physical rails.

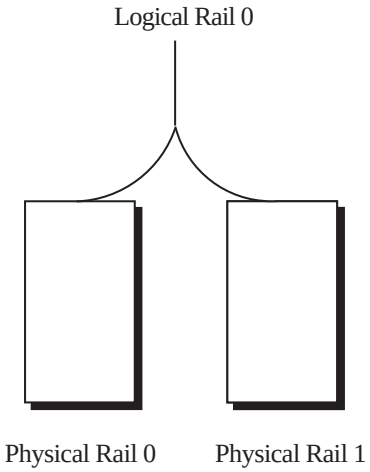
The failover pair style is the default for all multirail configurations. (It is the failover model used in previous TruCluster software releases.)

The order in which physical rails are paired into logical rails in a failover pair configuration is the order in which they are found at initialization: physical rails zero (0) and 1 are combined to give logical rail zero (0).

The failover pair configuration provides availability in the event of a physical rail failure, as the second physical rail is redundant. However, only the address space and bandwidth of a single physical rail are available at any given time.

Figure 1–2 shows a multirail MEMORY CHANNEL configuration in the failover pair style. The illustrated configuration has one logical rail, made up of two physical rails.

Figure 1–2: Failover Pair MEMORY CHANNEL Configuration



1.3.3 Configuring the MEMORY CHANNEL Multirail Model

When you implement the MEMORY CHANNEL multirail model, all nodes in a cluster must be configured with an equal number of physical rails, configured into an equal number of logical rails, each with the same failover style.

The first logical rail is numbered zero (0), the second logical rail is numbered 1, and so on, up to a maximum of four. This is represented by the constant `IMC_MAXRAILS` in the `imc.h` header file.

The system configuration parameter `rm_rail_style`, in the `/etc/sysconfigtab` file, is used to set multirail styles. The `rm_rail_style` parameter can be set to one of the following values:

- Zero (0) for a single-rail style
- 1 for a failover pair style

The default value of the `rm_rail_style` parameter is 1.

The `rm_rail_style` parameter must have the same value for all nodes in a cluster, or configuration errors will occur.

To change the value of the rm_rail_style parameter to zero (0) for a single-rail style, change the /etc/sysconfigtab file by adding or modifying the following stanza for the rm subsystem:

```
rm:
  rm_rail_style=0
```

Note

DIGITAL recommends that you use sysconfigdb(8) to modify or add to stanzas in the /etc/sysconfigtab file.

If you change the rm_rail_style parameter, you must halt the entire cluster, and then reboot each member system.

If the rm_rail_style parameter is set to 1 for a multirail configuration that has an odd number of physical rails, configuration errors will result.

Note

A cluster will fail if any logical rail fails. See Section 1.5.3 for more information.

Error handling for the MEMORY CHANNEL multirail model is implemented for specified logical rails. See Section 2.2.6 for a description of MEMORY CHANNEL API library error management functions and code examples.

Note

The MEMORY CHANNEL multirail model does not facilitate any type of cluster reconfiguration, such as the addition of hubs or MEMORY CHANNEL adapters. For such reconfiguration, you must first shut down the cluster completely.

1.4 Tuning Your MEMORY CHANNEL Configuration

The imc_init command initializes the MEMORY CHANNEL API library with certain resource defaults. Depending on your application, you may require more resources than the defaults allow. In some cases, you can change certain MEMORY CHANNEL parameters and virtual memory resource parameters to overcome these limitations. The following subsections describe these parameters and explain how to change them.

1.4.1 Extending MEMORY CHANNEL Address Space

The amount of total MEMORY CHANNEL address space that is available to the MEMORY CHANNEL API library is specified using the *maxalloc* parameter of the `imc_init` command. The maximum amount of MEMORY CHANNEL address space that can be attached for receive on a host is specified using the *maxrecv* parameter of the `imc_init` command. The default limit in each case is 10 MB. (Section 1.2 describes how to initialize the MEMORY CHANNEL API library using the `imc_init` command.)

You can use the `rcmgr`(8) command to change the value used during an automatic initialization by setting the variables `IMC_MAX_ALLOC` and `IMC_MAX_RECV`. For example, you can set the variables to allow a total of 80 MB of MEMORY CHANNEL address space to be made available to the MEMORY CHANNEL API library clusterwide, and to allow 60 MB of MEMORY CHANNEL address space to be attached for receive on the current host, as follows:

```
rcmgr set IMC_MAX_ALLOC 80
rcmgr set IMC_MAX_RECV 60
```

If you use the `rcmgr`(8) command to set new limits, they will take effect when the system reboots.

The MEMORY CHANNEL API library initialization command, `imc_init`, can be used to change both the amount of total MEMORY CHANNEL address space available and the maximum amount of MEMORY CHANNEL address space that can be attached for receive, after the MEMORY CHANNEL API library has been initialized. For example, to allow a total amount of 80 MB of MEMORY CHANNEL address space to be made available clusterwide, and to allow 60 MB of MEMORY CHANNEL address space to be attached for receive on the current host, use the following command:

```
imc_init -a 80 -r 60
```

If you use the `imc_init` command to set new limits, they will be lost when the system reboots, and the values of the `IMC_MAX_ALLOC` and `IMC_MAX_RECV` variables will be used as limits.

1.4.2 Increasing Wired Memory

Every page of MEMORY CHANNEL address space that is attached for receive must be backed by a page of physical memory on your system. This memory is nonpageable; that is, it is wired memory. The amount of wired memory on a host cannot be increased infinitely; the system configuration parameter `vm-syswiredpercent` will impose a limit. You can change the `vm-syswiredpercent` parameter in the `/etc/sysconfigtab` file.

For example, if you want to set the vm-syswiredpercent parameter to 80, the vm stanza in the /etc/sysconfigtab file must contain the following entry:

```
vm:
  vm-syswiredpercent=80
```

If you change the vm-syswiredpercent parameter, you must reboot the system.

Note

The default amount of wired memory is sufficient for most operations; DIGITAL recommends that you exercise caution in changing this limit.

1.4.3 Increasing Virtual Memory Map Entries

When a MEMORY CHANNEL region is attached or a lock is allocated, a virtual memory map entry is used. The number of virtual memory map entries is specified by the vm-mapentries parameter. The default value of the vm-mapentries parameter is 200 for DIGITAL UNIX Version 4.0D. If you attempt to attach many small regions, you may exceed the limit of virtual memory map entries; this is indicated by the IMC_MAPENTRIES error code.

You can change the vm-mapentries parameter in the /etc/sysconfigtab file. For example, if you want to set the vm-mapentries parameter to 300, the the vm stanza in the /etc/sysconfigtab file must contain the following entry:

```
vm:
  vm-mapentries=300
```

If you change the vm-mapentries parameter, you must reboot the system.

1.5 Troubleshooting

The following subsections describe error conditions that you may encounter when using the MEMORY CHANNEL API library functions, and suggest solutions.

1.5.1 IMC_NOTINIT Return Code

The IMC_NOTINIT status is returned when the imc_init command has not been run, or when the imc_init command has failed to run correctly.

The imc_init command must be run on each host in the MEMORY CHANNEL API cluster before you can use the MEMORY CHANNEL API library functions. (Section 1.2 describes how to initialize the MEMORY CHANNEL API library using the imc_init command.)

If the imc_init command does not run successfully, see Section 1.5.2 for suggested solutions.

1.5.2 MEMORY CHANNEL API Library Initialization Failure

The MEMORY CHANNEL API library may fail to initialize on a host; if this happens, an error message is displayed on the console, and written to the messages log file in the /usr/var/adm directory. Use the following list of error messages and solutions to eliminate the error:

- `MEMORY CHANNEL is not initialized for user access`

  This error message indicates that the current host has not been initialized to use the MEMORY CHANNEL API.

  To solve this problem, ensure that all MEMORY CHANNEL cables are correctly attached to the MEMORY CHANNEL adapters on this host. See Section 1.5.3 for more information on fatal errors caused by problems with the physical MEMORY CHANNEL configuration or interconnect.

- `MEMORY CHANNEL API - rm_no_inheritance must be 1`

  This error message indicates that the configurable attribute rm_no_inheritance is not set to 1. The installation procedure should automatically set the value of rm_no_inheritance to 1; however, if this value is changed after installation, the MEMORY CHANNEL API will fail to initialize.

  To solve this problem, the rm stanza in the /etc/sysconfigtab file must contain the following entry:

  ```
  rm:
    rm_no_inheritance=1
  ```

  If you change the rm_no_inheritance attribute, you must reboot the system.

- `MEMORY CHANNEL API - Number of logical rails 1 incompatible with rest of cluster 2`

  An error message of this form indicates that the number of logical rails on the booting node (in this example, the booting node has one logical rail) differs from the number of logical rails on the other nodes in the MEMORY CHANNEL API cluster (in this example, the other nodes have two logical rails).

To solve this problem, ensure that all MEMORY CHANNEL cables are correctly attached to the MEMORY CHANNEL adapters on this host. If this is not possible, because of missing or failed hardware, disconnect cables or power down hubs on other hosts so that they mirror this host.

- `MEMORY CHANNEL API - incompatible MEMORY CHANNEL Software Version 1.4`

  This error message indicates that a node in the MEMORY CHANNEL API cluster is running Version 1.4 or Version 1.4a of the TruCluster software.

  To solve this problem, upgrade all hosts to run Version 1.5 of the TruCluster software, and manually invoke the `imc_init` command.

- `MEMORY CHANNEL API - get_RM_information() failed with status 212`

  This error message indicates that logical rail zero (0) is inactive.

  To solve this problem, ensure that the hub is powered up and that all cables are connected properly. For more information on how to configure MEMORY CHANNEL hardware, see Chapter 5 of the TruCluster Software Products *Hardware Configuration* manual.

- `MEMORY CHANNEL API - insufficient wired memory`

  This error message indicates that the value of the `IMC_MAX_RECV` variable in the `/etc/config` file or the value of the `-r` option to the `imc_init` command is greater than the wired memory limit specified by the configuration parameter `vm-syswiredpercent`.

  To solve this problem, invoke the `imc_init` command with a smaller value for the *`maxrecv`* parameter, or increase the system wired memory limit as described in Section 1.4.2.

1.5.3 Fatal MEMORY CHANNEL Errors

Sometimes the MEMORY CHANNEL API will fail to initialize because of problems with the physical MEMORY CHANNEL configuration or interconnect. Error messages printed on the console in these circumstances do not mention the MEMORY CHANNEL API. The following subsections describe some of the more common reasons for such failures.

1.5.3.1 Logical Rail Failure

If any logical rail fails, a system panic occurs on one or more hosts in the cluster, and the following error message is displayed on the console:

```
panic (cpu 0): rm_delete_context: fatal MC error
```

To solve this problem, ensure that the hub is powered up and that all cables are connected properly; then halt the entire cluster, and reboot each member system.

1.5.3.2 Logical Rail Initialization Failure

If the logical rail configuration for a logical rail on this node does not match that of a logical rail on other cluster members, a system panic occurs on one or more hosts in the cluster, and error messages of the following form are displayed on the console:

```
rm_slave_init
rail configuration does not match cluster expectations for logical rail 0
logical rail 0 has failed initialization
rm_delete_context: lcsr = 0x2a80078, mcerr = 0x20001, mcport = 0x72400001
panic (cpu 0): rm_delete_context: fatal MC error
```

This error can occur if the configuration parameter `rm_rail_style` is not identical on every node.

To solve this problem, follow these steps:

1. Halt the system.
2. Boot `/genvmunix`.
3. Modify the `/etc/sysconfigtab` file as described in Section 1.3.3.
4. Reboot the kernel with MEMORY CHANNEL API cluster support (`/vmunix`).

1.5.3.3 MEMORY CHANNEL Cables Crossed

If some MEMORY CHANNEL cables are connected incorrectly, a system panic occurs on one or more hosts in the cluster, and error messages of the following form are displayed on the console:

```
rm_slave_init
slave unit boot phase 0: checking cables
rm_check_cables: cables crossed
logical rail 0 has failed initialization
rm_delete_context: lcsr = 0x2a80078, mcerr = 0x20001, mcport = 0x72400001
panic (cpu 0): rm_delete_context: fatal MC error
```

To solve this problem, connect the first MEMORY CHANNEL adapter installed in one system to the first adapter installed in the other system, connect the second adapter installed in one system to the second adapter installed in the other system, and so on. In standard hub mode, all MEMORY CHANNEL adapters on a system must be connected to the same slot position in each hub.

For more information on how to configure MEMORY CHANNEL hardware, see Chapter 5 of the TruCluster Software Products *Hardware Configuration* manual.

1.5.4 IMC_MCFULL Return Code

The IMC_MCFULL status is returned if there is not enough MEMORY CHANNEL address space to perform an operation.

The amount of total MEMORY CHANNEL address space available to the MEMORY CHANNEL API library is specified by using the *maxalloc* parameter of the imc_init command, as described in Section 1.5.2.

You can use the rcmgr(8) command, or the MEMORY CHANNEL API library initialization command, imc_init, to increase the amount of MEMORY CHANNEL address space that is available to the library clusterwide. See Section 1.4.1 for more details.

1.5.5 IMC_RXFULL Return Code

The IMC_RXFULL status is returned by the imc_asattach( ) function, if receive mapping space is exhausted when an attempt is made to attach a region for receive.

Note

The default amount of receive space on the current host is 10 MB.

The maximum amount of MEMORY CHANNEL address space that can be attached for receive on a host is specified using the *maxrecv* parameter of the imc_init command, as described in Section 1.2 .

You can use the rcmgr(8) command or the MEMORY CHANNEL API library initialization command, imc_init, to extend the maximum amount of MEMORY CHANNEL address space that can be attached for receive on the host. See Section 1.4.1 for more details.

1.5.6 IMC_WIRED_LIMIT Return Code

The IMC_WIRED_LIMIT return value indicates that an attempt has been made to exceed the maximum quantity of wired memory.

The system configuration parameter vm-syswiredpercent specifies the wired memory limit; see Section 1.4.2 for information on changing this limit.

1.5.7 IMC_MAPENTRIES Return Code

The IMC_MAPENTRIES return value indicates that the maximum number of virtual memory map entries has been exceeded for the current process.

The maximum number of virtual memory map entries is specified by the `vm-mapentries` parameter; see Section 1.4.3 for information on changing this limit.

1.5.8 IMC_NOMEM Return Code

The `IMC_NOMEM` return status indicates a `malloc` function failure while performing a MEMORY CHANNEL API function call.

This will happen if process virtual memory has been exceeded, and can be remedied by using the usual techniques for extending process virtual memory limits; that is, by using the `limit` command and the `unlimit` command for the C shell, and by using the `ulimit` command for the Bourne shell and the Korn shell.

1.5.9 IMC_NORESOURCES Return Code

The `IMC_NORESOURCES` return value indicates that there are insufficient MEMORY CHANNEL data structures available to perform the required operation. However, the amount of available MEMORY CHANNEL data structures is fixed, and cannot be increased by changing a parameter. To solve this problem, amend the application to use fewer regions or locks.

2

Application Notes

The MEMORY CHANNEL Application Programming Interface (API) implements highly efficient memory sharing between MEMORY CHANNEL API cluster members, with automatic error-handling, locking, and UNIX style protections. This chapter contains information to help you develop applications based on the MEMORY CHANNEL API library. It explains the differences between MEMORY CHANNEL address space and traditional shared memory, and describes how programming using MEMORY CHANNEL as a transport differs from programming using shared memory as a transport.

This chapter also contains examples that show how to use the MEMORY CHANNEL API library functions in programs. You will find these code files in the `/usr/examples/cluster/` directory. Each file contains compilation instructions.

The chapter discusses the following topics:

- Initializing the MEMORY CHANNEL API library for a user program (Section 2.1)
- Accessing MEMORY CHANNEL address space (Section 2.2)
- Clusterwide locks (Section 2.3)
- Cluster signals (Section 2.4)
- Cluster information (Section 2.5)
- Comparison of shared memory and message passing models (Section 2.6)

2.1 Initializing the MEMORY CHANNEL API Library for a User Program

The `imc_api_init( )` function is used to initialize the MEMORY CHANNEL API library in a user program. Call the `imc_api_init( )` function in a process before any of the other MEMORY CHANNEL API functions are called. If a process forks, the `imc_api_init( )` function must be called before calling any other API functions in the child process, or an undefined behavior will result.

2.2 Accessing MEMORY CHANNEL Address Space

The MEMORY CHANNEL interconnect provides a form of memory sharing between MEMORY CHANNEL API cluster members. The MEMORY CHANNEL API library is used to set up the memory sharing, allowing processes on different members of the cluster to exchange data using direct read and write operations to addresses in their virtual address space. When the memory sharing has been set up by the MEMORY CHANNEL API library, these direct read and write operations take place at hardware speeds without involving the operating system or the MEMORY CHANNEL API library software functions.

When a system is configured with MEMORY CHANNEL, part of the physical address space of the system is assigned to the MEMORY CHANNEL address space. The size of the MEMORY CHANNEL address space is specified by the `imc_init` command. A process accesses this MEMORY CHANNEL address space by using the MEMORY CHANNEL API to map a region of MEMORY CHANNEL address space to its own virtual address space.

Applications that wish to access the MEMORY CHANNEL address space on different cluster members can allocate part of the address space for a particular purpose by calling the `imc_asalloc( )` function. The *`key`* parameter associates a clusterwide key with the region. Other processes that allocate the same region also specify this key. This allows processes to coordinate access to the region.

To use an allocated region of MEMORY CHANNEL address space, a process maps the region into its own process virtual address space, using the `imc_asattach( )` function or the `imc_asattach_ptp( )` function. When a process attaches to a MEMORY CHANNEL region, an area of virtual address space the same size as the MEMORY CHANNEL region is added to the process virtual address space. When attaching the region, the process indicates whether the region is mapped to receive or transmit data, as follows:

- Transmit—Indicates that the region is to be used to transmit data on MEMORY CHANNEL. When a process writes to addresses in this virtual address region, the data is transmitted over the MEMORY CHANNEL interconnect to the other members of the MEMORY CHANNEL API cluster.

  To map a region for transmit, specify the value `IMC_TRANSMIT` for the *`dir`* parameter to the the `imc_asattach( )` function.

- Receive—Indicates that the region is to be used to receive data from MEMORY CHANNEL. In this case, the address space that is mapped into the process virtual address space is backed by a region of physical memory on the system. When data is transmitted on MEMORY CHANNEL, it is written into the physical memory of any hosts that have mapped the region for receive, so that processes on that system read

from the same area of physical memory. The process does not receive any data that is transmitted before the region is mapped.

To map a region for receive, use the value `IMC_RECEIVE` as the *`dir`* parameter for the `imc_asattach( )` function.

A process can attach to a MEMORY CHANNEL region in broadcast mode, point-to-point mode, or loopback mode. These methods of attach are described in Section 2.2.1.

Memory sharing using the MEMORY CHANNEL interconnect is similar to conventional shared memory in that, once it is established, simple accesses to virtual address space allow two different processes to share data. However, there are two differences between these memory sharing mechanisms that you must allow for, as follows:

- When conventional shared memory is created it is assigned a virtual address. In C programming terms, there is a pointer to the memory. This single pointer can be used both to read and write data to the shared memory. However, a MEMORY CHANNEL region can have two different virtual addresses assigned to it: a transmit virtual address and a receive virtual address. In C programming terms, there are two different pointers to manage; one pointer can only be used for write operations, the other pointer is used for read operations.
- In conventional shared memory, write operations are made directly to memory and are immediately visible to other processes reading from the same memory. However, when a write operation is made to a MEMORY CHANNEL region, the write operation is not made directly to memory but to the I/O system and the MEMORY CHANNEL hardware. This means that there is a delay before the data appears in memory on the receiving system. This is described in more detail in Section 2.2.5.

2.2.1 Attaching to MEMORY CHANNEL Address Space

The following subsections describe the ways in which a process can attach to MEMORY CHANNEL address space. There are three ways in which a process can attach to MEMORY CHANNEL address space, as follows:

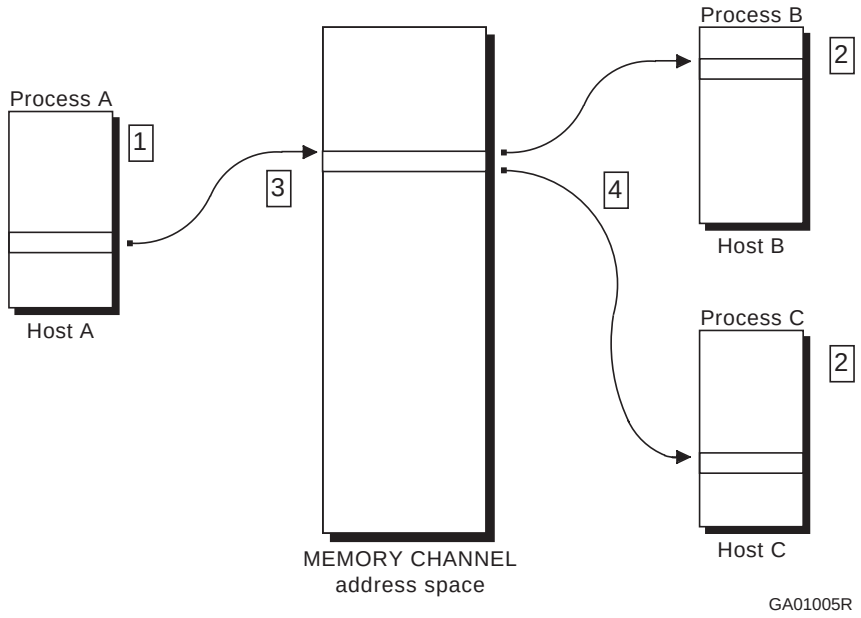
- Broadcast attach
- Point-to-point attach
- Loopback attach

This section also explains initial coherency, reading and writing MEMORY CHANNEL regions, latency related coherency, and error management, and includes some code examples.

2.2.1.1 Broadcast Attach

When one process maps a region for transmit, and other processes map the same region for receive, the data that the transmit process writes to the region is transmitted on MEMORY CHANNEL to the receive memory of the other processes. Figure 2–1 shows a three-host MEMORY CHANNEL implementation that shows how the address spaces are mapped.

Figure 2–1: Broadcast Address Space Mapping



With the address spaces mapped as shown in Figure 2–1, note the following:

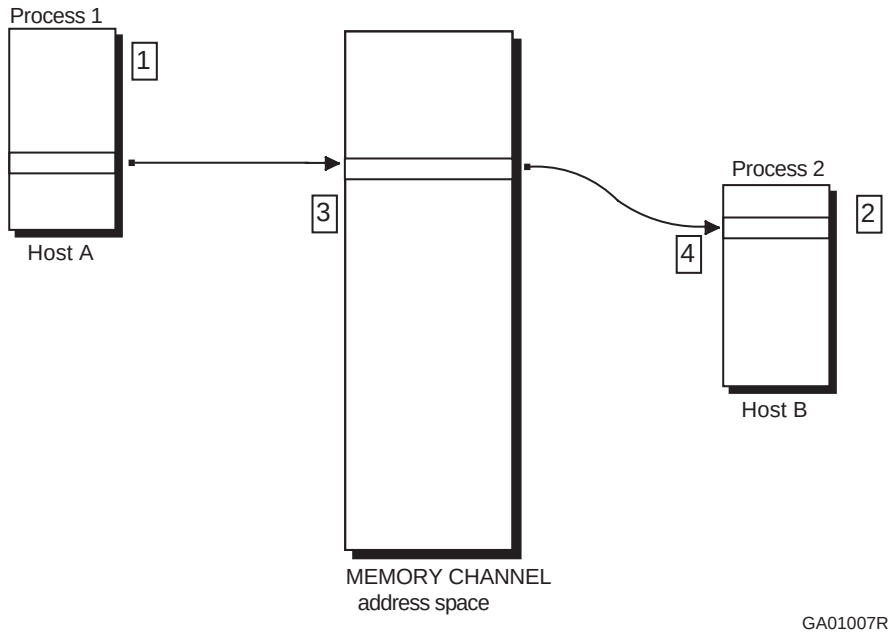
1. Process A allocates a region of MEMORY CHANNEL address space. Process A then maps the allocated region to its virtual address space when it attaches the region for transmit using the `imc_asattach()` function.
2. Process B and Process C both allocate the same region of MEMORY CHANNEL address space as Process A. However, unlike Process A, Process B and Process C both attach the region to receive data.
3. When data is written to the virtual address space of Process A, the data is transmitted on MEMORY CHANNEL.
4. When the data from Process A appears on MEMORY CHANNEL, it is written to the physical memory that backs the virtual address spaces of Process B and Process C.

2.2.1.2 Point-to-Point Attach

An allocated region of MEMORY CHANNEL address space can be attached for transmit in point-to-point mode to the virtual address space of a process on another node. This is done by calling the `imc_asattach_ptp( )` function with a specified host as a parameter. This means that writes to the region are sent only to the host specified in the parameter, and not to all hosts in the cluster.

Regions attached using the `imc_asattach_ptp( )` function are always attached in transmit mode, and are write-only. Figure 2–2 shows a two-host MEMORY CHANNEL implementation that shows point-to-point address space mapping.

Figure 2–2: Point-to-Point Address Space Mapping



With the address spaces mapped as shown in Figure 2–2, note the following:

1. Process 1 allocates a region of MEMORY CHANNEL address space. Process A then maps the allocated region to its virtual address space when it attaches the region point-to-point to Host B using the `imc_asattach_ptp( )` function.
2. Process 2 allocates the region and then attaches it for receive, using the `imc_asattach( )` function.

3. When data is written to the virtual address space of Process 1, the data is transmitted on MEMORY CHANNEL.
4. When the data from Process 1 appears on MEMORY CHANNEL, it is written to the physical memory that backs the virtual address space of Process 2 on Host B.

2.2.1.3 Loopback Attach

A region can be attached for both transmit and receive by processes on a host. Data written by the host is written to other hosts that have attached the region for receive. However, by default, data written by the host is not also written to the receive memory on that host; it is written only to other hosts. If you want a host to see data that it writes, you must specify the `IMC_LOOPBACK` flag to the `imc_asattach( )` function when attaching for transmit.

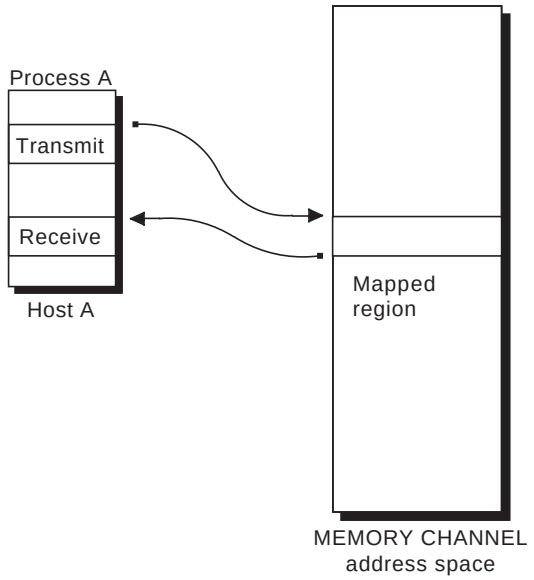
The loopback attribute of a region is set up on a per-host basis, and is determined by the value of the flag parameter to the first transmit attach on that host.

If you specify the value `IMC_LOOPBACK` for the *`flag`* parameter, two MEMORY CHANNEL transactions occur for every write, one to write the data and one to loop the data back.

Because of the nature of point-to-point attach mode, looped-back writes are not permitted.

Figure 2–3 shows a configuration in which a region of MEMORY CHANNEL address space is attached both for transmit with loopback and for receive.

Figure 2–3: Loopback Address Space Mapping



2.2.2 Initial Coherency

When a MEMORY CHANNEL region is attached for receive, the initial contents are undefined. This situation can arise because a process that has mapped the same MEMORY CHANNEL region for transmit might update the contents of the region before other processes map the region for receive. This is referred to as the initial coherency problem. You can overcome this in two ways:

- Write the application in a way that ensures that all processes attach the region for receive before any processes write to the region.
- At allocation time, specify that the region is coherent, by specifying the `IMC_COHERENT` flag when you allocate the region using the `imc_asalloc( )` function. This ensures that all processes will see every update to the region, regardless of when the processes attach the region.

  Coherent regions use the loopback feature. This means that two MEMORY CHANNEL transactions occur for every write, one to write the data and one to loop the data back; because of this, coherent regions have less available bandwidth than noncoherent regions.

2.2.3 Reading and Writing MEMORY CHANNEL Regions

Processes that attach a region of MEMORY CHANNEL address space can only write to a transmit pointer, and can only read from a receive pointer. Any attempt to read a transmit pointer will result in a segmentation violation.

Apart from explicit read operations on MEMORY CHANNEL transmit pointers, segmentation violations will also result from operations that cause the compiler to generate read-modify-write cycles; for example:

- Postincrement and postdecrement operations.
- Preincrement and predecrement operations.
- Assignment to simple data types that are not an integral multiple of four bytes.
- Use of the `bcopy`(3) library function where the length parameter is not an integral multiple of eight bytes, or where the source or destination arguments are not eight-byte aligned.
- Assignment to structures that are not quadword-aligned (that is, the value returned by the `sizeof( )` function is not an integral multiple of eight). This refers only to unit assignment of the whole structure; for example, `mystruct1 = mystruct2`.

2.2.4 Address Space Example

Example 2–1 shows how to initialize, allocate, and attach to a region of MEMORY CHANNEL address space, and also shows two of the differences between MEMORY CHANNEL address space and traditional shared memory:

- Initial coherency, as described in Section 2.2.2
- Asymmetry of receive and transmit regions, as described in Section 2.2.3

The sample program shown in Example 2–1 executes in master or slave mode, as specified by a command-line parameter. In master mode, the program writes its own process identifier (PID) to a data structure in the global MEMORY CHANNEL address space. In slave mode, the program polls a data structure in the MEMORY CHANNEL address space to determine the PID of the master process.

Note

Programs should be flexible in their use of keys, to prevent problems resulting from key clashes. The use of meaningful, application-specific keys is recommended.

Example 2–1: Accessing Regions of MEMORY CHANNEL Address Space

```
/* /usr/examples/cluster/mc_ex1.c */

#include <c_asm.h>
#include <sys/types.h>
#include <sys/imc.h>
#define VALID 756

main (int argc, char *argv[])
{
    imc_asid_t   glob_id;
    typedef struct {
                    pid_t     pid;
                    volatile int valid;                          1
                    } clust_pid;

    clust_pid   *global_record;
    caddr_t     add_rx_ptr = 0, add_tx_ptr = 0;
    int         status;
    int         master;
    int         logical_rail=0;

    /* check for correct number of arguments /*

    if (argc != 2) {
                    printf("usage: mcpid 0|1\n");
                    exit(-1);
                   }

    /* test if process is master or slave */

    master  = atoi(argv[1]);                                     2

    /* initialize MEMORY CHANNEL API library */

    status  = imc_api_init(NULL);                                3

    if (status < 0) {
                     imc_perror("imc_api_init::",status);        4
                     exit(-2);
                    }

    imc_asalloc(123, 8192, IMC_URW, 0, &glob_id,
                         logical_rail);                                5

    if (master) {
                 imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                              0, &add_tx_ptr);                   6

                 global_record = (clust_pid*)add_tx_ptr;         7
                 global_record->pid = getpid();
                 mb();                                           8
                 global_record->valid = VALID;
                 mb();
                }

    else        {  /* secondary process */

                 imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                              0, &add_rx_ptr);                   9

                 (char*)global_record  = add_rx_ptr;
```

Example 2–1: Accessing Regions of MEMORY CHANNEL Address Space (cont.)

```
                while ( global_record->valid != VALID)
                      ;  /* continue polling */                    10

                printf("pid of master process is %d\n",
                       global_record->pid);
               }
   imc_asdetach(glob_id);
   imc_asdealloc(glob_id);                                          11
}
```

1 The `valid` flag is declared as volatile to prevent the compiler from performing any optimizations that might prevent the code from reading the updated PID value from memory.

2 The first argument on the command line indicates whether the process is a master (argument equal to 1) or a slave process (argument not equal to 1).

3 The `imc_api_init( )` function initializes the MEMORY CHANNEL API library, and should be called before calling any of the other MEMORY CHANNEL API library functions.

4 All MEMORY CHANNEL API library functions return a zero (0) status if successful. The `imc_perror( )` function decodes error status values. For brevity, this example ignores the status from all functions other than the `imc_api_init( )` function.

5 The `imc_asalloc( )` function allocates a region of MEMORY CHANNEL address space with the following characteristics:

- `key=123`—The value identifies the region of MEMORY CHANNEL address space. Other applications that attach this region will use the same key value.
- `size=8192`—The size of the region is 8192 bytes.
- `perm=IMC_URW`—The access permission on the region is user read and write.
- `id=glob_id`—The `imc_asalloc( )` function returns this value, which uniquely identifies the allocated region. The program uses this value in subsequent calls to other MEMORY CHANNEL functions.
- `logical_rail=0`—The region is allocated using MEMORY CHANNEL logical rail zero (0).

6 The master process attaches the region for transmit by calling the `imc_asattach( )` function and specifying the `glob_id` identifier, which was returned by the call to the `imc_asalloc( )` function. The

imc_asattach( ) function returns add_tx_ptr, a pointer to the address of the region in the process virtual address space. The IMC_SHARED value signifies that the region is shareable, so other processes on this host can also attach the region.

7 The program overlays the global region with the global record structure and writes the process PID in the pid field of the global record. Note that the master process has attached the region for transmit; therefore, it can only write data in the field. An attempt to read the field will result in a segmentation violation; for example:

```
(pid_t)x = global_record->pid;
```

8 The program uses memory barrier instructions to ensure that the pid field is forced out of the Alpha CPU write buffer before the VALID flag is set.

9 The slave process attaches the region for receive by calling the imc_asattach( ) function and specifying the glob_id identifier, which was returned by the call to the imc_asalloc( ) function. The imc_asattach( ) function returns add_rx_ptr, a pointer to the address of the region in the process virtual address space. On mapping, the contents of the region may not be consistent on all processes that map the region. Therefore, start the slave process before the master to ensure that all writes by the master process appear in the virtual address space of the slave process.

10 The slave process overlays the region with the global record structure and polls the valid flag. The earlier declaration of the flag as volatile ensures that the flag is immune to compiler optimizations, which might result in the field being stored in a register. This ensures that the loop will load a new value from memory at each iteration and will eventually detect the transition to VALID.

11 At termination, the master and slave processes explicitly detach and deallocate the region by calling the imc_asdetach( ) function and the imc_asdealloc( ) function. In the case of abnormal termination, the allocated regions are automatically freed when the processes exit.

2.2.5 Latency Related Coherency

As described in Section 2.2.2, the initial coherency problem can be overcome by retransmitting the data after all mappings of the same region for receive have been completed, or by specifying at allocation time that the region is coherent. However, when a process writes to a transmit pointer, several microseconds can elapse before the update is reflected in the physical memory that corresponds to the receive pointer. If the process reads the receive pointer during that interval, the data it reads might be incorrect. This is known as the latency related coherency problem.

Latency problems do not arise in conventional shared memory systems. Memory and cache control ensure that store and load instructions are synchronized with data transfers.

Example 2–2 shows two versions of a program that decrements a global process count and detects the count reaching zero (0). The first program uses System V shared memory and interprocess communication. The second uses the MEMORY CHANNEL API library.

Example 2–2: System V IPC and MEMORY CHANNEL Code Comparison

```
/* /usr/examples/cluster/mc_ex2.c */

/*************************************
 *********  System V IPC example  *******
 *************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main()
{
   typedef struct {
                    int  proc_count;
                    int  remainder[2047]
                   } global_page;
   global_page  *mypage;
   int  shmid;

   shmid = shmget(123, 8192, IPC_CREAT |  SHM_R |  SHM_W);

   (caddr_t)mypage = shmat(shmid, 0,  0);  /* attach the
                                              global region     */

   mypage->proc_count ++;               /* increment process
                                           count                */

   /* body of program goes here */
           .
           .
           .
   /* clean up */

   mypage->proc_count --;               /* decrement process
                                           count                */
   if (mypage->proc_count == 0 )
                printf("The last process is exiting\n");
           .
           .
           .
}

/*************************************
 *******  MEMORY CHANNEL example  *******
 *************************************/

#include <sys/types.h>
#include <sys/imc.h>
main()
{
```

Example 2–2: System V IPC and MEMORY CHANNEL Code Comparison (cont.)

```
   typedef struct {
                   int  proc_count;
                   int  remainder[2047]
                  } global_page;
   global_page  *mypage_rx, *mypage_tx;                          1
   imc_asid_t   glob_id;
   int          logical_rail=0;
   int          temp;

   imc_api_init(NULL);

   imc_asalloc(123, 8192, IMC_URW | IMC_GRW, 0, &glob_id,
               logical_rail);                                        2

   imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                IMC_LOOPBACK, &(caddr_t)mypage_tx);              3

   imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                0, &(caddr_t)mypage_rx);                         4

   /* increment process count */

   mypage_tx->proc_count = mypage_rx->proc_count + 1;            5

   /* body of program goes here */
           .
           .
           .
   /* clean up */

   /* decrement process count

   temp = mypage_rx->proc_count - 1                              6

   mypage_tx->proc_count = temp;

   /* wait for MEMORY CHANNEL update to occur */

   while (mypage_rx->proc_count != temp)
          ;

   if (mypage_rx->proc_count == 0 )
                printf("The last process is exiting\n");
           .
           .
           .
}
```

1 The process must be able to read the data that it writes to the MEMORY CHANNEL global address space. Therefore, it declares two addresses, one for transmit and one for receive.

2 The `imc_asalloc( )` function allocates a region of MEMORY CHANNEL address space. The characteristics of the region are as follows:

- `key=123`—This value identifies the region of MEMORY CHANNEL address space. Other applications that attach this region will use the same key value.
- `size=8192`—The size of the region is 8192 bytes.
- `perm=IMC_URW | IMC_GRW`—The region is allocated with user and group read and write permission.
- `id=glob_id`—The `imc_asalloc( )` function returns this value, which uniquely identifies the allocated region. The program uses this value in subsequent calls to other MEMORY CHANNEL API library functions.
- `logical_rail=0`—The region is allocated using MEMORY CHANNEL logical rail zero (0).

3 This call to the `imc_asattach( )` function attaches the region for transmit at the address pointed to by the `mypage_tx` variable. The value of the `flag` parameter is set to `IMC_LOOPBACK`, so that any time the process writes data to the region, the data is looped back to the receive memory.

4 This call to the `imc_asattach( )` function attaches the region for receive at the address pointed to by the `mypage_rx` variable.

5 The program increments the global process count by adding 1 to the value in the receive pointer, and by assigning the result into the transmit pointer. When the program writes to the transmit pointer, it does not wait to ensure that the write instruction completes.

6 After the body of the program completes, the program decrements the process count and tests that the decremented value was transmitted to the other hosts in the cluster. To ensure that it examines the decremented count (rather than some transient value), the program stores the decremented count in a local variable, `temp`. It writes the decremented count to the transmit region, and then waits for the value in the receive region to match the value in `temp`. When the match occurs, the program knows that the decremented process count has been written to the MEMORY CHANNEL address space.

In this example, the use of the local variable ensures that the program compares the value in the receive memory with the value that was transmitted. An attempt to use the value in the receive memory before ensuring that the value had been updated may result in erroneous data being read.

2.2.6 Error Management

In a shared memory system, the process of reading and writing to memory is assumed to be error-free. In a MEMORY CHANNEL system, the error rate

is of the order of three errors per year. This is much lower than the error rates of standard networks and I/O subsystems.

The MEMORY CHANNEL hardware reports detected errors to the MEMORY CHANNEL software. The MEMORY CHANNEL hardware provides two guarantees that make it possible to develop applications that can cope with errors:

- It does not write corrupt data to host systems.
- It delivers data to the host systems in the sequence in which the data is written to the MEMORY CHANNEL hardware.

These guarantees simplify the process of developing reliable and efficient messaging systems.

The MEMORY CHANNEL API library provides the following functions to help applications implement error management:

- `imc_ckerrcnt_mr( )`—The `imc_ckerrcnt_mr( )` function checks for the existence of errors on a specified logical rail on MEMORY CHANNEL hosts. This allows transmitting processes to check whether or not errors occur when they send messages.
- `imc_rderrcnt_mr( )`—The `imc_rderrcnt_mr( )` function reads the clusterwide error count for the specified logical rail and returns the value to the calling program. This allows receiving processes to check the error status of messages that they receive.

The operating system maintains a count of the number of errors that occur on the cluster. The system increments the value whenever it detects a MEMORY CHANNEL hardware error in the cluster, and when a host joins or leaves the cluster.

The task of detecting and processing an error takes a small, but finite, amount of time. This means that the count returned by the `imc_rderrcnt_mr( )` function might not be up to date with respect to an error that has just occurred on another host in the cluster. On the local host, the count is always up to date.

Use the `imc_rderrcnt_mr( )` function to implement a simple and effective error-detection mechanism by reading the error count before transmitting a message, and including the count in the message. The receiving process compares the error count in the message body with the local value determined after the message arrives. The local value is guaranteed to be up to date, so if this value is the same as the transmitted value, then it is certain that no intervening errors occurred. Example 2–3 shows this technique.

Example 2–3: Error Detection Using the imc_rderrcnt_mr Function

```
/* /usr/examples/cluster/mc_ex3.c */

/*****************************************
*********  Transmitting Process  *********
*****************************************/

#include <sys/imc.h>
#include <c_asm.h>
main()
{
 typedef struct {
  volatile int  msg_arrived;
  int send_count;
  int remainder[2046];
 } global_page;
 global_page *mypage_rx, *mypage_tx;
 imc_asid_t glob_id;
 int  i;
 volatile int err_count;

 imc_api_init(NULL);

 imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
 imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
                      &(caddr_t)mypage_tx);
 imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                      &(caddr_t)mypage_rx);

 /*  save the error count  */
 while (  (err_count = imc_rderrcnt_mr(0) ) < 0 )
  ;

 mypage_tx->send_count = err_count;

 /* store message data */
 for (i = 0; i < 2046; i++)
  mypage_tx->remainder[i] = i;

 /* now mark as valid */
 mb();

 do {
  mypage_tx->msg_arrived = 1;
 } while (mypage_rx->msg_arrived != 1);  /* ensure no error on
                                                  valid flag         */

}


/*****************************************
***********  Receiving Process  **********
*****************************************/

#include <sys/imc.h>
main()
{
 typedef struct {
  volatile int  msg_arrived;
  int send_count;
  int remainder[2046];
```

Example 2–3: Error Detection Using the imc_rderrcnt_mr Function (cont.)

```
} global_page;
global_page *mypage_rx, *mypage_tx;
imc_asid_t glob_id;
int  i;
volatile int err_count;

imc_api_init(NULL);

imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                    &(caddr_t)mypage_rx);

/* wait for message arrival */
while ( mypage_rx->msg_arrived == 0 )
 ;

/* get  this systems error count */
while ( (err_count = imc_rderrcnt_mr(0) ) < 0 )
 ;

if (err_count == mypage_rx->send_count) {
 /* no error, process the body */
                           .....
}
else {
 /* do error  processing */
             ......
}
}
```

As shown in Example 2–3, the `imc_rderrcnt_mr( )` function can be safely used to detect errors at the receiving end of a message. However, it cannot be guaranteed to detect errors at the transmitting end. This is because there is a small, but finite, possibility that the transmitting process will read the error count before the transmitting host has been notified of an error occurring on the receiving host. In Example 2–3, the program must rely on a higher-level protocol informing the transmitting host of the error.

The `imc_ckerrcnt_mr( )` function provides guaranteed error detection for a specified logical rail. This function takes a user-supplied local error count and a logical rail number as parameters, and returns an error in the following circumstances:

- An outstanding error is detected on the specified logical rail
- Error processing is in progress
- The error count is higher than the supplied parameter

If the function returns successfully, no errors have been detected between when the local error count was stored and the `imc_ckerrcnt_mr( )` function was called.

The `imc_ckerrcnt_mr( )` function reads the MEMORY CHANNEL adapter hardware error status for the specified logical rail; this is a hardware operation that takes several microseconds. Therefore, the `imc_ckerrcnt_mr( )` function takes longer to execute than the `imc_rderrcnt_mr( )` function, which reads only a memory location.

Example 2–4 shows an amended version of the send sequence shown in Example 2–3. In Example 2–4, the transmitting process performs error detection.

Example 2–4: Error Detection Using the imc_ckerrcnt_mr Function

```
/* /usr/examples/cluster/mc_ex4.c */

/*******************************************/
/*  Transmitting Process With Error Detection */
/*******************************************/

#include <c_asm.h>
#define mb() asm("mb")

#include <sys/imc.h>
main()
{
 typedef struct {
  volatile int  msg_arrived;
  int send_count;
  int remainder[2046];
 } global_page;
 global_page *mypage_rx, *mypage_tx;
 imc_asid_t glob_id;
 int  i, status;
 volatile int err_count;

 imc_api_init(NULL);

 imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
 imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
                     &(caddr_t)mypage_tx);
 imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                     &(caddr_t)mypage_rx);

 /*  save the error count  */
 while (  (err_count = imc_rderrcnt_mr(0) ) < 0 )
  ;

 do {
  mypage_tx->send_count = err_count;

  /* store message data */
  for (i = 0; i < 2046; i++)
   mypage_tx->remainder[i] = i;

  /* now mark as valid */
  mb();

  mypage_tx->msg_arrived = 1;

        /*  if error occurs, retransmit */
```

Example 2–4: Error Detection Using the imc_ckerrcnt_mr Function (cont.)

```
 } while ( (status = imc_ckerrcnt_mr(&err_count,0)) != IMC_SUCCESS);
}
```

2.3 Clusterwide Locks

In a MEMORY CHANNEL system, the processes communicate by reading and writing regions of the MEMORY CHANNEL address space. The preceding sections contain sample programs that show arbitrary reading and writing of regions. In practice, however, a locking mechanism is sometimes needed to provide controlled access to regions and to other clusterwide resources. The MEMORY CHANNEL API library provides a set of lock functions that enable applications to implement access control on resources.

The MEMORY CHANNEL API library implements locks by using mapped pages of the global MEMORY CHANNEL address space. For efficiency reasons, locks are allocated in sets rather than individually. The `imc_lkalloc( )` function allows you to allocate a lock set. For example, if you want to use 20 locks, it is more efficient to create one set with 20 locks than five sets with four locks each, and so on.

To facilitate the initial coordination of distributed applications, the `imc_lkalloc( )` function allows a process to atomically (that is, in a single operation) allocate the lock set and acquire the first lock in the set. This feature allows the process to determine whether or not it is the first process to allocate the lock set. If it is, the process is guaranteed access to the lock and can safely initialize the resource.

Instead of allocating the lock set and acquiring the first lock atomically, a process could call the `imc_lkalloc( )` function and then the `imc_lkacquire( )` function. In that case, however, there is a risk that another process might acquire the lock between the two function calls, and the first process would not be guaranteed access to the lock.

Example 2–5 shows a program in which the first process to lock a region of MEMORY CHANNEL address space initializes the region, and the processes that subsequently access the region simply update the process count.

Example 2–5: Locking MEMORY CHANNEL Regions

```
/* /usr/examples/cluster/mc_ex5.c */

#include <sys/types.h>
#include <sys/imc.h>
```

Example 2–5: Locking MEMORY CHANNEL Regions (cont.)

```
main ( )
{
    imc_asid_t   glob_id;
    imc_lkid_t   lock_id;
    int          locks = 4;
    int          status;

    typedef struct {
                    int    proc_count;
                    int    pattern[2047];
                   } clust_rec;

    clust_rec *global_record_tx, *global_record_rx;             1
    caddr_t      add_rx_ptr = 0, add_tx_ptr = 0;
    int     j;

    status  = imc_api_init(NULL);

    imc_asalloc(123, 8192, IMC_URW, 0, &glob_id, 0);

    imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                 IMC_LOOPBACK, &add_tx_ptr);

    imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                 0, &add_rx_ptr);

    global_record_tx = (clust_rec*) add_tx_ptr;                 2
    global_record_rx = (clust_rec*) add_rx_ptr;

    status = imc_lkalloc(456, &locks, IMC_LKU,  IMC_CREATOR,
                         &lock_id);                             3
    if (status == IMC_SUCCESS)
    {

    /* This is the first process. Initialize the global region  */

        global_record_tx->proc_count = 0;                          4
        for (j = 0; j < 2047; j++)
                global_record_tx->pattern[j] = j;

        /* release the lock */
        imc_lkrelease(lock_id, 0);                              5

    }

   /* This is a secondary process */

    else if (status == IMC_EXISTS)
    {
       imc_lkalloc(456, &locks, IMC_LKU, 0, &lock_id);          6

       imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT);              7

       /* wait for access to region */

       global_record_tx->proc_count = global_record_rx->proc_count+1; 8

       /* release the lock */
```

Example 2–5: Locking MEMORY CHANNEL Regions (cont.)

```
       imc_lkrelease(lock_id, 0);

    }

    /* body of program goes here */
           .
           .
           .
    /* clean up */

    imc_lkdealloc(lock_id);                                    9
    imc_asdetach(glob_id);
    imc_asdealloc(glob_id);
}
```

1 The process, in order to read the data that it writes to the MEMORY CHANNEL global address space, maps the region for transmit and for receive. See Example 2–2 for a detailed description of this procedure.

2 The program overlays the transmit and receive pointers with the global record structure.

3 The process tries to create a lock set that contains four locks and a `key` value of `456`. The call to the `imc_lkalloc( )` function also specifies the `IMC_CREATOR` flag. Therefore, if the lock set is not already allocated, the function will automatically acquire lock zero (0). If the lock set already exists, the `imc_lkalloc( )` function fails to allocate the lock set and returns the value `IMC_EXISTS`.

4 The process that creates the lock set (and consequently holds lock zero (0)) initializes the global region.

5 When the process finishes initializing the region, it calls the `imc_lkrelease( )` function to release the lock.

6 Secondary processes that execute after the region has been initialized, having failed in the first call to the `imc_lkalloc( )` function, now call the function again, without the `IMC_CREATOR` flag. Because the value of the `key` parameter is the same (`456`), this call allocates the same lock set.

7 The secondary process calls the `imc_lkacquire( )` function to acquire lock zero (0) from the lock set.

8 The secondary process updates the process count and writes it to the transmit region.

9 At the end of the program, the processes release all MEMORY CHANNEL resources.

When a process acquires a lock, other processes executing on the cluster cannot acquire that lock.

Waiting for locks to become free entails busy spinning and has a significant effect on performance. Therefore, in the interest of overall system performance, applications should acquire locks only as they are needed and release them promptly.

2.4 Cluster Signals

The MEMORY CHANNEL API library provides the `imc_kill( )` function to allow processes to send signals to specified processes executing on a remote host in a cluster. This function is similar to the UNIX `kill`(2) function. The main difference is that the `imc_kill( )` function does not support the sending of signals to multiple processes.

2.5 Cluster Information

The following sections discuss how to use the MEMORY CHANNEL API functions to access cluster information, and how to access status information from the command line.

2.5.1 Using MEMORY CHANNEL API Functions to Access MEMORY CHANNEL API Cluster Information

The MEMORY CHANNEL API library provides the `imc_getclusterinfo( )` function, which allows processes to get information about the hosts in a MEMORY CHANNEL API cluster. The function returns one or more of the following:

- A count of the number of hosts in the cluster, and the name of each host.
- The number of logical rails in the cluster.
- The active MEMORY CHANNEL logical rails bitmask, with a bit set for each active logical rail.

The function does not return information about a host unless the MEMORY CHANNEL API library is initialized on the host.

The MEMORY CHANNEL API library provides the `imc_wait_cluster_event( )` function to block a calling thread until a specified cluster event occurs. The following MEMORY CHANNEL API cluster events are valid:

- A host joins or leaves the cluster.
- The logical rail configuration of the cluster changes.

The `imc_wait_cluster_event( )` function checks the current representation of the MEMORY CHANNEL API cluster configuration item

being monitored and returns the new MEMORY CHANNEL API cluster configuration.

Example 2–6 shows how you can use the `imc_getclusterinfo( )` function with the `imc_wait_cluster_event( )` function to request the names of the members of the MEMORY CHANNEL API cluster and the active MEMORY CHANNEL logical rails bitmask, and then wait for an event change on either.

Example 2–6: Requesting MEMORY CHANNEL API Cluster Information; Waiting for MEMORY CHANNEL API Cluster Events

```
/* /usr/examples/cluster/mc_ex6.c */

#include <sys/imc.h>

main ( )
{

    imc_railinfo    mask;
    imc_hostinfo    hostinfo;

    int             status;
    imc_infoType    items[3];
    imc_eventType   events[3];


    items[0] = IMC_GET_ACTIVERAILS;
    items[1] = IMC_GET_HOSTS;
    items[2] = 0;

    events[0] = IMC_CC_EVENT_RAIL;
    events[1] = IMC_CC_EVENT_HOST;
    events[2] = 0;

    imc_api_init(NULL);

    status = imc_getclusterinfo(items,2,mask,sizeof(imc_railinfo),
                                &hostinfo,sizeof(imc_hostinfo));

    if (status != IMC_SUCCESS)
        imc_perror("imc_getclusterinfo:",status);

    status = imc_wait_cluster_event(events, 2, 0,
                                mask, sizeof(imc_railinfo),
                                &hostinfo, sizeof(imc_hostinfo));

    if ((status != IMC_HOST_CHANGE) && (status != IMC_RAIL_CHANGE))
        imc_perror("imc_wait_cluster_event didn't complete:",status);

}   /*main*/
```

2.5.2 Accessing MEMORY CHANNEL Status Information from the Command Line

The MEMORY CHANNEL API library provides the `imcs` command to report on MEMORY CHANNEL status. The `imcs` command writes information to the standard output about currently active MEMORY CHANNEL facilities. The output is displayed as a list of regions or lock sets, and includes the following information:

- The type of subsystem that created the region or lock set (possible values are IMC or PVM)
- An identifier for the MEMORY CHANNEL region
- An application-specific key that refers to the MEMORY CHANNEL region or lock set
- The size, in bytes, of the region
- The access mode of the region or lock set
- The username of the owner of the region or lock set
- The group of the owner of the region or lock set
- The MEMORY CHANNEL logical rail used for the region
- A flag specifying the coherency of the region
- The number of locks available in the lock set

2.6 Comparison of Shared Memory and Message Passing Models

There are two models that you can use to develop applications based on the MEMORY CHANNEL API library:

- Shared memory
- Message passing

At first, the shared memory approach might seem more suited to the MEMORY CHANNEL features. However, developers who use this model must deal with the latency, coherency, and error-detection problems described in this chapter. In some cases, it might be more appropriate to develop a simple message-passing library that hides these problems from applications. The data transfer functions in such a library could be implemented completely in user space. Therefore, they would operate as efficiently as implementations based on the shared memory model.

3

MEMORY CHANNEL API Library Interface

This chapter describes the functions that are provided by the MEMORY CHANNEL API library application programming interface (API). It discusses the following topics:

- Header files (Section 3.1)
- Library (Section 3.2)
- Compiling applications that use the MEMORY CHANNEL API library (Section 3.3)
- Overview of MEMORY CHANNEL API library commands and functions (Section 3.4)
- Command descriptions (Section 3.5)
- Function descriptions (Section 3.6)

The descriptions of the MEMORY CHANNEL API library functions are presented in alphabetical order and in reference page style.

3.1 Header Files

The MEMORY CHANNEL API library includes the `imc.h` header file. This file defines the data structures, data types, and constants associated with the MEMORY CHANNEL API library, including a definition for the version of MEMORY CHANNEL software. The header file is called `imc.h` and is located in the `/usr/include/sys` directory. Use the following line to include the header file in programs that use the MEMORY CHANNEL API library:

```
#include <sys/imc.h>
```

3.2 Library

The MEMORY CHANNEL API library functions are located in the system library. The shared version is located in the `/usr/shlib` directory; it is called `libimc.so`. The nonshared version is located in the `/usr/ccs/lib` directory; it is called `libimc.a`.

3.3 Compiling Applications that Use the MEMORY CHANNEL API Library

Use the cc command to compile applications based on the MEMORY CHANNEL API library, making sure that you include the library. The following example shows how to compile a program called *program.c:*

```
cc -o program program.c -limc
```

3.4 Overview of MEMORY CHANNEL API Library Commands and Functions

This section contains reference information that introduces the MEMORY CHANNEL API library commands and functions.

NAME

imc – Introduction to the MEMORY CHANNEL Application Programming Interface (API)

DESCRIPTION

The MEMORY CHANNEL Application Programming Interface (API) library provides user space access to the MEMORY CHANNEL services available in the TruCluster environment.

MEMORY CHANNEL API library functions provide the following services:

- MEMORY CHANNEL API cluster information
- Access to MEMORY CHANNEL address space
- A clusterwide lock system
- MEMORY CHANNEL API cluster signals
- MEMORY CHANNEL API library management

Commands

The following MEMORY CHANNEL API library commands are available:

imc_init(1)	Initializes and configures the MEMORY CHANNEL API library on the current host.
imcs(1)	Reports on MEMORY CHANNEL status.

Functions

MEMORY CHANNEL API functions can be grouped into categories. An introduction to each category and a brief description of each function follows.

MEMORY CHANNEL API Cluster Information

A MEMORY CHANNEL API cluster is formed when a number of hosts are physically connected by a MEMORY CHANNEL interconnect, and when each host has invoked the imc_init(1) command.

Independent MEMORY CHANNEL interconnects, or physical rails, can be configured as logical rails, in one of the following styles:

- Single-rail

  This configuration has a one-to-one relationship between a physical rail and a logical rail, with no failover properties.

- Failover pair

  In this configuration, a logical rail consists of two physical rails, with one physical rail inactive and available on standby in case the active physical rail fails. Failover is transparent to the user.

The following functions provide information about the MEMORY CHANNEL API cluster:

`imc_getclusterinfo`(3)

Gets information about the hosts and the logical rails that form a MEMORY CHANNEL API cluster.

`imc_wait_cluster_event`(3)

Blocks the caller until a MEMORY CHANNEL API cluster event occurs. MEMORY CHANNEL API cluster events include hosts entering the MEMORY CHANNEL API cluster or leaving the MEMORY CHANNEL API cluster, and logical rails coming on line or going off line.

Accessing MEMORY CHANNEL Address Space

A process accesses MEMORY CHANNEL address space by mapping a region of the address space into its own process virtual address space. This is done by allocating a region and then attaching the allocated region to the virtual address space of a process.

By attaching the same region to the virtual address space of two different processes, it is possible for one process to write data into the virtual address space of the other process using standard store and load instructions.

The following functions are available to allow access to MEMORY CHANNEL address space:

imc_asalloc(3)	Allocates a region of MEMORY CHANNEL address space on a specified logical rail.
imc_asattach(3)	Attaches a region of MEMORY CHANNEL address space to the virtual address space of a process.
imc_asattach_ptp(3)	Attaches in point-to-point mode a region of MEMORY CHANNEL address space to the virtual address space of a process.
imc_bcopy(3)	Provides an efficient way of copying data into MEMORY CHANNEL address space.
imc_asdetach(3)	Detaches a region of MEMORY CHANNEL address space from the virtual address space of the calling process.
imc_dealloc(3)	Deallocates a region of MEMORY CHANNEL address space.

The MEMORY CHANNEL hardware takes care of all error detection. The MEMORY CHANNEL API provides the following routines to access the error state of the hardware:

imc_ckerrcnt_mr(3)	Checks for the existence of outstanding errors on a specified logical rail on MEMORY CHANNEL hosts.
imc_rderrcnt_mr(3)	Reads the clusterwide error count for a specified logical rail.

Clusterwide Lock System

The MEMORY CHANNEL API provides a clusterwide lock facility. Locks are allocated in sets; they are not allocated individually. Clusterwide locks are managed using the following functions:

imc_lkalloc(3)	Creates a lock set.
imc_lkacquire(3)	Acquires a lock from a lock set.

`imc_lkrelease`(3)	Releases a lock from a lock set.
`imc_lkdealloc`(3)	Deallocates a lock set.

MEMORY CHANNEL API Cluster Signals

The MEMORY CHANNEL API allows processes to send signals to processes executing on other hosts in the MEMORY CHANNEL API cluster, using the following function:

`imc_kill`(3)	Sends a signal to a running process.

MEMORY CHANNEL API Management

The following MEMORY CHANNEL API management functions are available:

`imc_api_init`(3)	Initializes the MEMORY CHANNEL API library.

Note

The `imc_api_init` function must be called before any other MEMORY CHANNEL API function is called.

`imc_perror`(3)	Prints a message that explains a MEMORY CHANNEL function error.

3.5 Command Descriptions

This section contains reference information for the MEMORY CHANNEL API library initialization command, and the MEMORY CHANNEL API library status report command.

imc_init(1)

NAME

imc_init – Initializes and configures the MEMORY CHANNEL API library on the current host

SYNOPSIS

/usr/sbin/imc_init [-a *maxalloc*] [-r *maxrecv*]

OPTIONS

-a *maxalloc*	Specifies, in MB, the total amount of MEMORY CHANNEL address space to be made available to the MEMORY CHANNEL API library. The default amount of address space is 10 MB. This is a clusterwide limit.
-r *maxrecv*	Specifies, in MB, the maximum amount of MEMORY CHANNEL address space that can be attached for receive on the host. The default amount of receive space is 10 MB. This limit applies only to the current host.

DESCRIPTION

The imc_init command, available in a Production Server or MEMORY CHANNEL software configuration, initializes and configures the MEMORY CHANNEL API library on the current host. Initialization of the MEMORY CHANNEL API library occurs either by automatic execution of the imc_init command at system boot time, or by the system administrator invoking the command from the command line after the system boots.

Initialization of the MEMORY CHANNEL API library at system boot time is controlled by the IMC_AUTO_INIT variable in the /etc/rc.config file. If the value of this variable is set to 1, the imc_init command is invoked at system boot time. When the MEMORY CHANNEL API library is initialized at boot time, the values of the -a *maxalloc* and -a *maxrecv* flags are set to the values specified by the IMC_MAX_ALLOC and IMC_MAX_RECV variables in the /etc/rc.config file.

If the IMC_AUTO_INIT variable is set to zero (0), the MEMORY CHANNEL API library is not initialized at system boot time. The system administrator must invoke the imc_init command to initialize the library. The parameter values in the /etc/rc.config file are not used when the imc_init command is manually invoked.

The imc_init command initializes the MEMORY CHANNEL API library the first time it is invoked, whether this happens at system boot time or after the system has booted. The value of the -a *maxalloc* flag must be the same on all hosts in the MEMORY CHANNEL API cluster. If different values are specified, the maximum value specified for any host determines the clusterwide value that applies to all hosts.

After the MEMORY CHANNEL API library has initialized on the current host, the system administrator can invoke the imc_init command again to reconfigure the values of the *maxalloc* and *maxrecv* resource limits, without forcing a reboot. The system administrator can increase or decrease either limit, but the new limits cannot be lower than the current usage of the resources. Reconfiguring the MEMORY CHANNEL API cluster from the command line does not read or modify the values specified in the /etc/rc.config file. The system administrator can use the rcmgr(8) command to modify the parameters and have them take effect when the system reboots.

You must have root privileges to execute the imc_init command.

ERROR MESSAGES

The imc_init command prints the following error messages:

- `Receive area is bigger than the maximum allocation`

  The receive size is larger than the maximum allocation size.

- `No MEMORY CHANNEL memory available`

  There is not enough MEMORY CHANNEL address space to initialize the MEMORY CHANNEL API library.

- `No MEMORY CHANNEL resources available`

  There are insufficient MEMORY CHANNEL data structures available to initialize the MEMORY CHANNEL API library.

- `MEMORY CHANNEL is not initialized for user access`

This host has not been initialized to use the MEMORY CHANNEL API. Ensure that the MEMORY CHANNEL cables are properly connected.

- `Privileged command`

  You do not have root privileges. You must have root privileges to execute the `imc_init` command.

- `System wired memory limit cannot be exceeded. See kernel vm parameter vm-syswiredpercent`

  An attempt has been made to exceed the maximum quantity of system wired memory. The amount of MEMORY CHANNEL address space that can be attached for receive on the host cannot be increased beyond the limit imposed by the system parameter *vm-syswiredpercent*.

- `Invalid parameter specification`

  An attempt has been made to set the *maxalloc* parameter or the *maxrecv* parameter to zero (0), or to a non-numeric or a negative value.

FILES

`/usr/sbin/imc_init`	Specifies the command path.
`/etc/rc.config`	Contains the variables that control whether or not the MEMORY CHANNEL API library is initialized at system boot time, and specifies the parameter values to be applied on initialization.

SEE ALSO

Introduction: `imc`(3)

Commands: `rcmgr`(8), `imcs`(1)

NAME

imcs – Reports on MEMORY CHANNEL status

SYNOPSIS

/usr/sbin/imcs [-m] [-l] [-r] [-f] [-h]

OPTIONS

`-m`	Displays the names of all hosts that have initialized the MEMORY CHANNEL API.
`-l`	Displays information about all lock sets in use in the MEMORY CHANNEL API cluster.
`-r`	Displays information about all allocated regions in the MEMORY CHANNEL API cluster.
`-f`	Displays full MEMORY CHANNEL status information.
`-h`	Displays a user help message on the `imcs` command.

DESCRIPTION

The `imcs` command writes information to the standard output about currently active MEMORY CHANNEL facilities. The details displayed vary according to the flags used with the command. If no flags are specified, the `imcs` command displays the names of all MEMORY CHANNEL API cluster members, and information about active MEMORY CHANNEL regions and MEMORY CHANNEL lock sets.

The information is displayed as a list of regions or lock sets under the following headings:

`Type` — The type of the subsystem that created the region. Possible values are:

`IMC`	Region was created using MEMORY CHANNEL API.
`PVM`	Region was created using DIGITAL Parallel Virtual Machine (PVM).

`ID` — An identifier that uniquely identifies the MEMORY CHANNEL region.

`KEY` — An application-specific key that refers to the MEMORY CHANNEL region.

`SIZE` — The size, in bytes, of the MEMORY CHANNEL address space region allocated.

`MODE` — The access mode of the region or lock set. The access mode consists of nine bits and is similar to the UNIX permission convention, except that there is no execute bit. There are three sets of three bits each. The first set of three bits refers to the owner's permissions; the next set refers to permissions of others in the user group of the region; and the last set refers to all other permissions.

Within each set of three bits, the first character indicates permission to read the region, the second character indicates permission to write to the region, and the last character is currently unused. The permissions are indicated as follows:

`r`	Read permission is granted
`w`	Write permission is granted

-	This character is not used at present
OWNER	The username of the owner of the region or lock set.
GROUP	The group of the owner of the region or lock set.
RAIL	Specifies the MEMORY CHANNEL logical rail used for the region. The first logical rail is numbered zero (0), the second logical rail is numbered 1, and so on, up to a maximum defined by a constant, IMC_MAXRAILS.
FLAG	Specifies whether an allocated region was created as coherent, point-to-point, point-to-point coherent, or non-coherent. Flags apply only to MEMORY CHANNEL regions, not to locks.
NLOCKS	The number of locks available in the lock set.

EXAMPLES

When the /usr/sbin/imcs command is entered with no flags specified, details of MEMORY CHANNEL API cluster members, active MEMORY CHANNEL regions, and MEMORY CHANNEL lock sets are displayed, as shown in the following example:

```
MEMORY CHANNEL Cluster Members:
        member1.mydmn.myorg
        member2.mydmn.myorg
        member3.mydmn.myorg

MEMORY CHANNEL Regions:

Type  ID     KEY            SIZE       MODE       OWNER     GROUP
IMC   1740   5634309        204800     rw-rw-rw-  user1     users
IMC   1686   5634307        204800     rw-rw-rw-  user2     users
IMC   1627   5634306        483228     rw-rw-rw-  user2     users
IMC   1626   5634305        8192       rw-rw-rw-  user3     users
PVM   1576   17231442411520 393216     rw-rw-rw-  user4     users
IMC   612    611            4382720    r--------  user5     users

MEMORY CHANNEL Lock Sets:
```

```
Type  KEY            MODE       OWNER     GROUP     NLOCKS
IMC   37231          rw-rw-rw-  user1     users     110
PVM   17231442345984 rw-rw-rw-  user2     users     2
IMC   612            ------rw-  root      users     4
```

When the -f flag is used with the /usr/sbin/imcs command, the logical rail used is displayed under the RAIL heading, and the coherency of each allocated region is displayed under the FLAG heading, in addition to the details displayed in the previous example. Possible values under the FLAG heading are:

- Coherent
- Point-to-point
- Point-to-point coherent
- Non-coherent

SEE ALSO

Introduction: imc(3)

Command: imc_init(1)

Functions: imc_api_init(3), imc_asalloc(3), imc_asattach(3), imc_asattach_ptp(3), imc_lkacquire(3), imc_lkalloc(3)

3.6 Function Descriptions

This section contains reference information for each of the MEMORY CHANNEL API library functions.

imc_api_init(3)

NAME

`imc_api_init( )` – Initializes the MEMORY CHANNEL API library

SYNOPSIS

```
#include <sys/imc.h>
int imc_api_init(
        unsigned long* i_param);
```

PARAMETERS

i_param	This parameter is reserved for future use by DIGITAL. You must set the value of this parameter to NULL.

DESCRIPTION

The `imc_api_init( )` function initializes the MEMORY CHANNEL API library. A process must call the `imc_api_init( )` function before calling any of the other MEMORY CHANNEL API functions.

If a process forks, the `imc_api_init( )` function must be called before calling any other API functions in the child process, or undefined behavior will result.

RETURN VALUES

The `imc_api_init` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_NOTINIT`	This host has not been initialized to use the MEMORY CHANNEL API library.
`IMC_NORESOURCES`	There are insufficient MEMORY CHANNEL data structures available to initialize the MEMORY CHANNEL API library.
`IMC_INITERR`	An error occurred while initializing the MEMORY CHANNEL API environment.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

NAME

`imc_asalloc( )` – Allocates a region of MEMORY CHANNEL address space on a specified logical rail

SYNOPSIS

```
#include <sys/imc.h>
int imc_asalloc(
          imc_key_t key,
          imc_size_t size,
          imc_perm_t perm,
          int flag,
          imc_asid_t* id,
          int logical_rail);
```

PARAMETERS

key
Specifies an application-specific key that refers to the region. Other processes that allocate the region also specify this key. This ensures that access to the region is coordinated on a clusterwide basis.

size
Specifies the size, in bytes, of the MEMORY CHANNEL address space region to be allocated. The `imc_asalloc( )`function allocates address space, in pages, and it rounds up the value specified by the *size* parameter accordingly.

perm
Specifies the read and write permissions for the allocated region. The permission code is similar to the UNIX permission convention, except that there is no execute flag. The value of the *perm* parameter is obtained by carrying out a logical OR operation on the following values:

`IMC_URW`	User read and write
`IMC_UR`	User read
`IMC_UW`	User write
`IMC_GRW`	Group read and write

IMC_GR	Group read
IMC_GW	Group write
IMC_ORW	Other read and write
IMC_OR	Other read
IMC_OW	Other write

flag

Specifies whether or not the region is to be mapped into all hosts on the MEMORY CHANNEL API cluster at the time of allocation. The *flag* parameter has the following values:

IMC_COHERENT	Allocate a coherent region. When other processes allocate and attach this region, they will see all updates to the region since the region was created. When the IMC_COHERENT flag is specified, the physical pages that underlie the region are nonpageable on all hosts.
ZERO (0)	Do not allocate a coherent region. When processes on other hosts allocate and attach the region, they will see all updates to the region from then on. However, the initial contents of the region are indeterminate.

id

Returns an identifier that uniquely identifies the allocated MEMORY CHANNEL region.

logical_rail

Specifies the MEMORY CHANNEL logical rail to use. The first logical rail is numbered zero (0), the

second logical rail is numbered 1, and so on, up to a maximum defined by a constant, IMC_MAXRAILS.

DESCRIPTION

The imc_asalloc( ) function allocates a region of MEMORY CHANNEL address space on a specified logical rail. If the function successfully allocates the region, it returns the region identifier in the *id* parameter. If the function call is unsuccessful, the value of the *id* parameter is undefined. If a region with the key specified in the *key* parameter has already been allocated on the MEMORY CHANNEL API cluster, then the imc_asalloc( ) function returns the identifier of that region, and does not allocate a new region. Individual applications should define their own naming scheme for keys. The use of meaningful application-specific keys is recommended.

It is possible for multiple processes on a given host to allocate the same region of MEMORY CHANNEL address space. When multiple processes allocate a region, the permissions specified by each process must be compatible clusterwide; otherwise, an error condition will result.

The permissions associated with a region are determined by the first process to allocate the region.

When processes on multiple hosts allocate a region, the initial contents of the region might not be the same on all hosts. This situation can arise because a process that has mapped the same MEMORY CHANNEL region for transmit might update the contents of the region before other processes map the region for receive. To ensure that the region is coherent on all hosts in the MEMORY CHANNEL API cluster, specify the IMC_COHERENT flag when allocating the region.

The MEMORY CHANNEL API library maintains the total amount of available MEMORY CHANNEL address space as a clusterwide resource. If the imc_asalloc( )function tries to allocate a region that exceeds the amount of address space available, an error condition will result.

RETURN VALUES

The imc_asalloc function returns one of the following values:

IMC_SUCCESS	Normal successful completion.

IMC_BADPARM	An invalid parameter was specified in the call to the imc_asalloc( )function.
IMC_BADRAIL	The logical rail number specified in the call to the imc_asalloc( ) function is invalid, or the logical rail is inactive.
IMC_BADSIZE	The specified region is already allocated, and the size of the region as specified in this call to the imc_asalloc( )function does not match the size specified in the previous call.
IMC_COHERENCYERR	The specified region is already allocated, and the value of the *flag* parameter IMC_COHERENT specified in this call to the imc_asalloc( ) function does not match the value specified in the previous call.
IMC_MCFULL	There is not enough MEMORY CHANNEL address space to allocate the amount specified by the *size* parameter.
IMC_NOMEM	There is insufficient local memory available to allocate the region.
IMC_NORESOURCES	There are insufficient MEMORY CHANNEL data structures available to allocate the region.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.
IMC_PERMIT	The specified region is already allocated, with a permission code that is incompatible with the code specified in the *perm* parameter.
IMC_PRIOR	The region has already been allocated by this process.

IMC_WRONGRAIL	The specified region has already been allocated by a process on the MEMORY CHANNEL API cluster, using the same key, on a specific logical rail; the specified region cannot now be allocated on a different logical rail.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_asattach(3), imc_asattach_ptp(3), imc_asdealloc(3), imc_asdetach(3), imc_bcopy(3)

NAME

`imc_asattach` – Attaches an allocated region of MEMORY CHANNEL address space to the virtual address space of a process

SYNOPSIS

```
#include <sys/imc.h>
int imc_asattach(
          imc_asid_t id,
          imc_dir_t dir,
          int mode,
          int flag,
          caddr_t* address);
```

PARAMETERS

id

Identifies the region of MEMORY CHANNEL address space to be attached. The identifier specified by the *id* parameter must have previously been generated by a call to the `imc_asalloc` function.

dir

Specifies whether the region is attached to transfer data to the MEMORY CHANNEL address space or to receive data from it. The *dir* parameter contains one of the following values:

`IMC_TRANSMIT`	Attach the region as a transmit area.
`IMC_RECEIVE`	Attach the region as a receive area.

mode

Specifies the sharing mode, shared or nonshared, for the region. If the region is designated as shared, multiple processes executing on a given host can attach the region to their process virtual address space. The sharing mode is specified by the first process on the host to attach the region. The sharing mode is host-specific. Other processes that subsequently attach the region cannot change the sharing mode. If a calling process tries to attach a region that has an incompatible sharing mode, an error condition will result.

The *mode* parameter has the following values:

IMC_SHARED	The region is shared.
IMC_NONSHARED	The region is not shared.

flag

Specifies, for a transmit region, that all writes to the region are looped back to the host that writes the data; or, for a receive region, that a user-supplied address will be specified in the *address* parameter.

If this flag is not set for a transmit region, processes on this host that attach the region for receive will not see the data that is transmitted from the host. The flag to enable the loopback feature is set by the first process on the host to attach the transmit region. Subsequent calls to the imc_asattach( ) function on the same host must adhere to the convention established by the first call to the function.

For transmit attaches, you must enable the loopback feature when attaching to coherent regions. (A coherent region is one for which the IMC_COHERENT flag is specified in the imc_asalloc( ) function call that allocates the region.)

The *flag* parameter has one of the following values for a transmit attach:

IMC_LOOPBACK	Enable the loopback feature.
ZERO (0)	Disable the loopback feature.

For receive attaches, use the *flag* parameter to attach to a user-supplied address. The *flag* parameter has one of the following values for attach to a user-supplied address:

IMC_USE_ADDR	Attach to the address specified by the user in the *address* parameter.
ZERO (0)	Attach to an address in the process virtual address space assigned by the kernel, and return that address in the *address* parameter.

address

For transmit attaches, returns the address in the process virtual address space that is mapped to the attached region of MEMORY CHANNEL address space. This address is assigned by the kernel. This also applies to receive attaches where the *flag* parameter has the value ZERO (0).

For receive attaches, if the *flag* parameter has the value IMC_USE_ADDR, the address must be user-specified in the *address* parameter. The address must be page-aligned, and must represent a hole in the process virtual address space. Also, the extent of the hole must be enough to contain the region.

DESCRIPTION

The imc_asattach( ) function attaches a region of MEMORY CHANNEL address space to an address in the virtual address space of the calling process. The region must first have been allocated by means of a call to the imc_asalloc( ) function.

The calling process uses the *dir* parameter to attach the region for receive or transmit. Transmit regions are attached as write-only. Any attempt to read a transmit region will result in a segmentation violation. Therefore, some C operations, such as postincrement and predecrement, will cause a segmentation violation. Accesses to storage locations that are not integral multiples of four bytes will generate read-modify-write cycles that will also cause segmentation violations. Library functions such as bcopy(3) will induce this behavior when the *length* parameter is not an integral

number of eight bytes, or when the source or destination arguments are not eight-byte aligned. The imc_bcopy( ) function is designed to be used instead of the bcopy(3) function in such cases, as its *src* parameter and its *dest* parameter can both have an arbitrary alignment.

Attaching a region to receive data does not guarantee that the contents of the region are the same as for other processes attached to the region. Any previous writes to the region are not reflected in the process address space, but subsequent writes do appear. To ensure that the contents of the region are the same for all processes, specify the IMC_COHERENT flag in the imc_asalloc( ) function when allocating the region. Otherwise, the process must use application-specific mechanisms to transmit any existing memory content to the new region.

RETURN VALUES

The imc_asattach function returns one of the following values:

IMC_SUCCESS	Normal successful completion.
IMC_BADADDR	In the case of a receive attach, the *flag* parameter has the value IMC_USE_ADDR, and an invalid address was specified in the *address* parameter.
IMC_BADPARM	An invalid parameter was specified in the call to the imc_asattach( )function.
IMC_BADREGION	The region specified in the call to the imc_asattach( )function is invalid.
IMC_LATEJOIN	This host joined the MEMORY CHANNEL API cluster after the region was allocated.
IMC_LOOPBACKERR	Another process on this host has already attached the region, specifying a different value for the *flag* parameter than the value specified in this call to the imc_asattach( ) function; or the value of the *flag* parameter is incorrect. (If the IMC_COHERENT flag is specified when the imc_asalloc( ) function allocates the region, the IMC_LOOPBACK flag must

	be specified in the call to the `imc_asattach( )` function.)
`IMC_MAPENTRIES`	An attempt has been made to exceed the maximum number of process map entries. This maximum is set by the *`vm_mapentries`* parameter
`IMC_MCFULL`	There is not enough MEMORY CHANNEL address space to attach to a coherent region.
`IMC_NOMAPPER`	Attach to a coherent region could not be completed because the `imc_mapper` daemon was not found on a host in the MEMORY CHANNEL API cluster.
`IMC_NONSHARERR`	The region has already been mapped as nonshared; it cannot now be mapped as shared.
`IMC_NORESOURCES`	There are insufficient MEMORY CHANNEL data structures available to attach the region.
`IMC_NOTALLOC`	The region is not allocated.
`IMC_NOTINIT`	This host has not been initialized to use the MEMORY CHANNEL API library.
`IMC_PERMIT`	The process is not permitted to attach the region.
`IMC_PTPERR`	An attempt was made to attach for transmit to a region already in use as a point-to-point attach region; or an attempt was made to attach for receive, on a host other than the targeted host, to a point-to-point attach region.
`IMC_RECMAPPED`	The region has already been mapped by the process to receive data.

IMC_RXFULL	There are no more pages of physical memory available to the MEMORY CHANNEL API library.
IMC_SHARERR	The region has already been mapped as shared; it cannot now be mapped as nonshared.
IMC_XMITMAPPED	The region has already been mapped by the process to transmit data.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_asalloc(3), imc_asattach_ptp(3), imc_asdealloc(3), imc_asdetach(3), imc_bcopy(3)

NAME

`imc_asattach_ptp` – Attaches in point-to-point mode an allocated region of MEMORY CHANNEL address space to the virtual address space of a process

SYNOPSIS

```
#include <sys/imc.h>

int imc_asattach_ptp(
        imc_asid_t id,
        int mode,
        int flag,
        char* hostname,
        caddr_t* address);
```

PARAMETERS

id

Identifies the region of MEMORY CHANNEL address space to be attached. The identifier specified by the *id* parameter must have previously been generated by a call to the `imc_asalloc` function.

mode

Specifies the sharing mode, shared or nonshared, for the region. If the region is designated as shared, multiple processes executing on a given host can attach the region to their process virtual address space. The sharing mode is specified by the first process on the host to attach the region. The sharing mode is host-specific. Other processes that subsequently attach the region cannot change the sharing mode. If a calling process tries to attach a region that has an incompatible sharing mode, an error condition will result.

The *mode* parameter has the following values:

`IMC_SHARED`	The region is shared.
`IMC_NONSHARED`	The region is not shared.

flag

The loopback feature is not permitted for point-to-point regions.

The *flag* parameter has the following value for a transmit attach:

ZERO (0)	Disable the loopback feature.

hostname — Specifies the name of the host to which the region is attached for point-to-point transmission.

address — Returns the address in the process virtual address space that is mapped to the attached region of MEMORY CHANNEL address space. This address is assigned by the kernel.

DESCRIPTION

The imc_asattach_ptp( ) function attaches a region of MEMORY CHANNEL address space to an address in the virtual address space of the calling process. The region must first have been allocated by means of a call to the imc_asalloc( ) function.

The imc_asattach_ptp( ) function attaches the region in point-to-point mode. This means that writes to the region are sent only to the host specified in the *hostname* parameter. In contrast, writes to regions attached by means of a call to the imc_asattach( ) function are broadcast to all hosts in the MEMORY CHANNEL API cluster.

Regions attached using the imc_asattach_ptp( ) function are always attached in transmit mode.

Because of the nature of point-to-point attach mode, looped-back writes are not permitted.

RETURN VALUES

The imc_asattach_ptp function returns one of the following values:

IMC_SUCCESS	Normal successful completion.
IMC_BADPARM	An invalid parameter was specified in the call to the imc_asattach_ptp( ) function.

IMC_BADREGION	The region specified in the call to the `imc_asattach_ptp( )` function is invalid.
IMC_LOOPBACKERR	Another process on this host has already attached the region, specifying a different value for the *flag* parameter than the value specified in this call to the `imc_asattach_ptp( )` function; or the value of the *flag* parameter is incorrect. You cannot enable the loopback feature when calling the `imc_asattach_ptp( )` function.
IMC_LATEJOIN	This host joined the MEMORY CHANNEL API cluster after the region was allocated.
IMC_MAPENTRIES	An attempt has been made to exceed the maximum number of process map entries. This maximum is set by the *vm_mapentries* parameter.
IMC_MCFULL	There is not enough MEMORY CHANNEL address space to attach to a coherent region.
IMC_NOMAPPER	Attach to a coherent region could not be completed because the `imc_mapper` daemon was not found on a host in the MEMORY CHANNEL API cluster.
IMC_NONSHARERR	The region has already been mapped as nonshared; it cannot now be mapped as shared.
IMC_NORESOURCES	There are insufficient MEMORY CHANNEL data structures available to attach the region.
IMC_NOTALLOC	The region is not allocated.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.

IMC_PERMIT	The process is not permitted to attach the region.
IMC_PTPERR	This value is returned if one of the following events occurs: • An attempt is made to attach a region already attached in point-to-point mode to a different target host. • An attempt is made to point-to-point attach a region that is already broadcast attached (attached for read on more than one host). • An attempt is made to point-to-point attach a region that is already broadcast transmit attached. • An attempt is made to point-to-point attach a region that is already attached for read on the local host.
IMC_SHARERR	The region has already been mapped as shared; it cannot now be mapped as nonshared.
IMC_XMITMAPPED	The region has already been mapped by the process to transmit data.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_asalloc(3), imc_asattach(3), imc_asdealloc(3), imc_asdetach(3), imc_bcopy(3), imc_getclusterinfo(3)

NAME

`imc_asdealloc` – Deallocates a region of MEMORY CHANNEL address space

SYNOPSIS

```
#include <sys/imc.h>
int imc_asdealloc(
        imc_asid_t id);
```

PARAMETER

id	Identifies the region of MEMORY CHANNEL address space to be deallocated. The identifier specified by the *id* parameter must have previously been generated by a call to the `imc_asalloc( )`function.

DESCRIPTION

The `imc_asdealloc( )` function deallocates a region of MEMORY CHANNEL address space. Mapped regions must be detached by means of a call to the `imc_asdetach( )` function before being deallocated; otherwise, an error condition will result.

Deallocating a region will not necessarily free the region of MEMORY CHANNEL address space. This is because multiple processes can allocate a given region; the space is freed only when the last process on the MEMORY CHANNEL API cluster deallocates the region.

All MEMORY CHANNEL regions allocated by a process are automatically deallocated when the process exits.

RETURN VALUES

The `imc_asdealloc` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_ATTACHED`	The specified region of MEMORY CHANNEL address space is attached by the process. The region must be detached before it can be deallocated.

IMC_BADPARM	An invalid parameter was specified in the call to the imc_asdealloc( )function.
IMC_BADREGION	The region specified by the *id* parameter was not found.
IMC_NOTALLOC	The region is not allocated.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_asalloc(3), imc_asattach(3), imc_asattach_ptp(3), imc_asdetach(3)

NAME

`imc_asdetach` – Detaches a region of MEMORY CHANNEL address space from the virtual address space of the calling process

SYNOPSIS

```
#include <sys/imc.h>
int imc_asdetach(
          imc_asid_t id);
```

PARAMETER

id	Identifies the region of MEMORY CHANNEL address space to be detached. The identifier specified by the *id* parameter must be the one generated by the call to the `imc_asalloc( )` function that allocated the region.

DESCRIPTION

The `imc_asdetach( )` function detaches a region of MEMORY CHANNEL address space. When the function is called, it detaches all transmit and receive regions associated with the identifier specified by the *id* parameter. After a region is detached, all addresses associated with the region become invalid.

All MEMORY CHANNEL regions attached by a process are automatically detached when the process exits.

RETURN VALUES

The `imc_asdetach` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_BADPARM`	An invalid parameter was specified in the call to the `imc_asdetach( )` function.
`IMC_BADREGION`	The region specified in the call to the `imc_asdetach( )` function is invalid.

IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_asalloc(3), imc_asattach(3), imc_asattach_ptp(3), imc_asdealloc(3)

NAME

imc_bcopy – Efficient data copy to a MEMORY CHANNEL transmit region

SYNOPSIS

```
#include <sys/imc.h>
long imc_bcopy(
         void *src,
         void *dest,
         long length,
         long dest_write_only,
         long first_dest_quad);
```

PARAMETERS

src	Points to the source data buffer for the imc_bcopy function.
dest	Points to the destination data buffer for the imc_bcopy function.
length	Specifies the length, in bytes, of the original data buffer.
dest_write_only	Specifies whether the destination is a write-only pointer.
first_dest_quad	Specifies the contents of the first quadword of the destination.

DESCRIPTION

The imc_bcopy( ) function copies *length* bytes from the buffer pointed to by the *src* parameter into the buffer pointed to by the *dest* parameter.

The imc_bcopy( ) function is highly optimized for the Alpha architecture and implements an extremely efficient copy operation. You can use the imc_bcopy( ) function for a high-bandwidth copy between two buffers in normal memory, as well as for copying to MEMORY CHANNEL transmit addresses, regardless of buffer alignment or data length.

imc_bcopy(3)

A MEMORY CHANNEL region may be attached for transmit (that is, for write) using the `imc_asattach( )` function or the `imc_asattach_ptp( )` function. The address for such a region is write-only, and any attempt to read from a transmit address will result in a segmentation violation. In addition, segmentation violations will result from any operation that causes the compiler to generate read-modify-write cycles. For example:

- Assignment to simple data types that are not an integral multiple of four bytes.
- Use of the `bcopy`(3) function where the *length* parameter is not an integral multiple of eight bytes, or where the source or destination arguments are not eight-byte aligned.

The `imc_bcopy( )` function is designed to be used instead of the `bcopy`(3) function in such cases, as its *src* parameter and its *dest* parameter can both have an arbitrary alignment.

If the value of the *dest_write_only* parameter is zero (0), unaligned writes to the *dest* address can cause the quadwords containing the first and last destination bytes to be read.

If the value of the *dest_write_only* parameter is nonzero, as it would be for MEMORY CHANNEL transmit addresses, the *first_dest_quad* parameter value is used as the contents of the first quadword of the destination, and zero (0) is used as the contents of the last quadword of the destination. If the caller does not know the contents of the first quadword of the destination, use zero (0) as the value of the *first_dest_quad* parameter. This will result in up to three bytes of zeros before the start of the copied data, and up to three bytes of zeros after the end of the copied data.

The `imc_bcopy( )` function returns the last quadword written to the destination. You can use this capability to concatenate several noncontiguous buffers to a contiguous write-only destination. To perform this operation, known as a gather operation, use the return value from one call to the `imc_bcopy( )` function as the *first_dest_quad* parameter for the next call to the `imc_bcopy( )` function. If you are not performing a gather operation, that is, if the start of the *dest* parameter is not in the same quadword as the end of the previous *dest* parameter, then the value of the *first_dest_quad* parameter should be zero.

RESTRICTIONS

If the source and destination buffers overlap, the result of the copy operation is undefined.

EXAMPLES

1. This example shows how to use the `imc_bcopy( )` function to copy between two buffers that have arbitrary alignment. The destination buffer is not a MEMORY CHANNEL transmit address. In this example, 25 bytes are copied from an aligned source to a destination that is not aligned on a quadword boundary.

```
int        source[256];
char       destination[1024];
long       last_quad;

/* fill in source buffer */

/* copy part of source buffer */

last_quad = imc_bcopy(source,destination+3,25,0,0);
```

2. This example shows how to use the `imc_bcopy( )` function to copy data to a MEMORY CHANNEL transmit address. In the example, 18 bytes are copied to a MEMORY CHANNEL transmit address at an offset of 12 bytes from the beginning of the region.

```
int        source[256];
caddr_t    tx_addr;
imc_asid_t id;
int        status;
int        prev_err;
long       last_quad;

/* allocate and attach destination buffer */

status = imc_api_init(NULL);
if (status != IMC_SUCCESS)
    imc_perror("imc_api_init",status);

/* allocate a region of size 8K using key 678 on logical rail zero */
status = imc_asalloc(678,8192,IMC_URW,0,&id,0);
if (status != IMC_SUCCESS)
    imc_perror("imc_asalloc",status);

/* attach for transmit without LOOPBACK */
```

```
status = imc_asattach(id,IMC_TRANSMIT,IMC_SHARED,0,&tx_addr);
if (status != IMC_SUCCESS)
    imc_perror("imc_asattach",status);

/* fill in source buffer */

/* copy part of the source buffer and check for errors */

do {
    prev_err = imc_rderrcnt_mr(0);
    last_quad = imc_bcopy(source,tx_addr+12,18,1,0);
} while ((status = imc_ckerrcnt_mr(&prev_err,0)) != IMC_SUCCESS);
```

3. This example shows how to use the imc_bcopy( ) function to copy data to a MEMORY CHANNEL transmit address from several sources. The sources may be noncontiguous.

```
int        *src1;
long       *src2;
char       *src3;
long       len1,len2,len3;
caddr_t    tx_addr;
int        status;
int        prev_err;
long       last_quad;

/* allocate and attach destination buffer */

/* assign and fill source buffers and their lengths */

/* append the source buffers at their destination */

do {
    prev_err = imc_rderrcnt_mr(0);
    last_quad = imc_bcopy(src1,tx_addr,len1,1,0);
    last_quad = imc_bcopy(src2,tx_addr+len1,len2,1,last_quad);
    last_quad = imc_bcopy(src3,tx_addr+len1+len2,len3,1,
                                                    last_quad);
} while ((status = imc_ckerrcnt_mr(&prev_err,0)) != IMC_SUCCESS);
```

RETURN VALUES

The imc_bcopy( ) function returns the last quadword written to the destination buffer.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

imc_bcopy(3)

Functions: `imc_asalloc`(3), `imc_asattach`(3), `imc_asattach_ptp`(3)

NAME

`imc_ckerrcnt` – Checks for the existence of outstanding errors on MEMORY CHANNEL hosts in a MEMORY CHANNEL API cluster

SYNOPSIS

```
#include <sys/imc.h>
int imc_ckerrcnt(
        int *errcnt);
```

PARAMETER

errcnt	Specifies the current process error count across all logical rails, and returns the updated error count.

DESCRIPTION

Note

DIGITAL recommends using the `imc_ckerrcnt_mr( )` function rather than the `imc_ckerrcnt( )` function.

The `imc_ckerrcnt( )` function checks for the existence of outstanding errors across all logical rails on the other MEMORY CHANNEL hosts in a MEMORY CHANNEL API cluster. It returns the `IMC_MC_ERROR` value if any of the following conditions apply:

- The function detects an outstanding error on another host in the MEMORY CHANNEL API cluster.
- The function detects that error handling is in progress.
- The total error count on all logical rails is greater than the value supplied in the *errcnt* parameter.

If an error count is being updated at the time the `imc_ckerrcnt( )` function is called, the function returns a negative value in the *errcnt* parameter. Programs should check for this eventuality and call the function again to ensure that the error has been handled.

You can use the imc_ckerrcnt( ) function along with the imc_rderrcnt( ) function to construct application-specific error-detection protocols.

RETURN VALUES

The imc_ckerrcnt function returns one of the following values:

IMC_SUCCESS	Normal successful completion: no MEMORY CHANNEL errors detected.
IMC_MC_ERROR	A MEMORY CHANNEL error was detected.
IMC_BADPARM	An invalid parameter was specified in the call to the imc_ckerrcnt function.
IMC_INITERR	A fatal error occurred while initializing the error-checking mechanism.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.
IMC_NORESOURCES	There are insufficient MEMORY CHANNEL data structures available to perform the operation.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_ckerrcnt_mr(3), imc_perror(3), imc_rderrcnt(3), imc_rderrcnt_mr(3)

NAME

`imc_ckerrcnt_mr` – Checks for the existence of outstanding errors on a specified logical rail on MEMORY CHANNEL hosts in a MEMORY CHANNEL API cluster

SYNOPSIS

```
#include <sys/imc.h>
int imc_ckerrcnt_mr(
        int *errcnt,
        int logical_rail);
```

PARAMETER

errcnt	Specifies the current process error count on the specified logical rail, and returns the updated error count.
logical_rail	Specifies the MEMORY CHANNEL logical rail that is to be checked for errors. The first logical rail is numbered zero (0), the second logical rail is numbered 1, and so on, up to a maximum defined by a constant, `IMC_MAXRAILS`.

DESCRIPTION

The `imc_ckerrcnt_mr( )` function checks for the existence of outstanding errors on the specified MEMORY CHANNEL logical rail. It returns the `IMC_MC_ERROR` value if any of the following conditions apply:

- The function detects an outstanding error on the specified logical rail.
- The function detects that error handling is in progress.
- The error count is greater than the value supplied in the *errcnt* parameter.

If an error count is being updated at the time the `imc_ckerrcnt_mr( )` function is called, the function returns a negative value in the *errcnt* parameter. Programs should check for this eventuality and call the function again to ensure that the error has been handled.

You can use the imc_ckerrcnt_mr( ) function along with the imc_rderrcnt_mr( ) function to construct application-specific error detection protocols.

RETURN VALUES

The imc_ckerrcnt_mr function returns one of the following values:

IMC_SUCCESS	Normal successful completion: no MEMORY CHANNEL errors detected.
IMC_MC_ERROR	A MEMORY CHANNEL error was detected on the specified logical rail; or error handling is in progress; or the error count is greater than the value supplied in the *errcnt* parameter.
IMC_BADPARM	An invalid parameter was specified in the call to the imc_ckerrcnt_mr function.
IMC_BADRAIL	The logical rail number specified in the call to the imc_ckerrcnt_mr function is invalid; or the logical rail is inactive.
IMC_INITERR	A fatal error occurred while initializing the error-checking mechanism.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.
IMC_NORESOURCES	There are insufficient MEMORY CHANNEL data structures available to complete the operation.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_perror(3), imc_rderrcnt_mr(3)

NAME

`imc_getclusterinfo` – Gets information about the hosts participating in a MEMORY CHANNEL API cluster

SYNOPSIS

```
#include <sys/imc.h>
int imc_getclusterinfo(
        imc_infotype *i_items,
        int i_nitems,
        [,char *io_data,
        int i_datalen] ...);
```

PARAMETERS

`i_items`
Points to an array that contains the enumerated type of each item to be returned. The last element of the array must be zero (0). Valid types are:

`IMC_GET_HOSTS`	Returns information on the number of hosts in a MEMORY CHANNEL API cluster and the name of each host, in a data structure of type `imc_hostinfo`.
`IMC_GET_NRAILS`	Returns the number of logical rails in the MEMORY CHANNEL API cluster, in a variable of type `unsigned int`.
`IMC_GET_ACTIVERAILS`	Returns the logical rail numbers of the active logical rails in the MEMORY CHANNEL API cluster, in a variable of type `imc_railinfo`.

`i_nitems`
Specifies the number of items in the array *`i_items`*.

io_data — Points to a buffer that contains the item of MEMORY CHANNEL API cluster information requested.

i_datalen — Specifies the length of the buffer identified by the *io_data* parameter.

DESCRIPTION

The imc_getclusterinfo( ) function returns information on items in a MEMORY CHANNEL API cluster. One or more of the following items may be requested:

- A count of the number of hosts participating in the MEMORY CHANNEL API cluster, and the name of each host.
- The number of logical rails in the MEMORY CHANNEL API cluster.
- The active MEMORY CHANNEL logical rails bitmask, which contains the numbers of the active logical rails.

A request of zero (0) items is valid and will return nothing.

The request items are returned in data structures, as follows:

imc_hostinfo — The data structure of type *imc_hostinfo* contains the following fields:

name[IMC_MAXHOSTS][MAXHOSTNAMELEN]

The host names are returned in the two-dimensional *name* array. The string containing the host name is zero-terminated. The elements in the array are numbered zero (0) to *(num-1)*.

num — The number of hosts in the MEMORY CHANNEL API cluster is returned in the *num* field.

imc_railinfo The active MEMORY CHANNEL logical rails bitmask is returned in the *imc_railinfo* array (with bit zero (0) representing logical rail number zero (0), bit 1 representing logical rail number 1, and so on).

Note

The imc_getclusterinfo( ) function lists only those hosts that have initialized the MEMORY CHANNEL API library.

EXAMPLES

1. The following program extract requests the names of the members of the MEMORY CHANNEL API cluster (functionality that was previously provided by the imc_gethosts function, which is now obsolete):

```
imc_hostinfo    hostinfo;
int             status,i;
imc_infoType    items[2];

items[0] = IMC_GET_HOSTS;
items[1] = 0;

status =
    imc_getclusterinfo(items,1,&hostinfo,sizeof(imc_hostinfo));

if (status != IMC_SUCCESS)
     imc_perror("imc_getclusterinfo:",status);
 else
     for (i=0; i<hostinfo.num; i++)
         printf("Member: %s\n",hostinfo.name[i]);
```

2. The following program extract requests the active MEMORY CHANNEL logical rails bitmask and prints out the numbers of the active logical rails:

```
imc_railinfo    mask;
int             status,i;
imc_infoType    items[2];

items[0] = IMC_GET_ACTIVERAILS;
items[1] = 0;

status = imc_getclusterinfo(items,1,mask,sizeof(imc_railinfo));
```

```
if (status != IMC_SUCCESS)
     imc_perror("imc_getclusterinfo:",status);
 else
     for (i=0; i<IMC_MAXRAILS;i++)
         if (IMC_IS_RAIL_ACTIVE(mask,i))
             printf("Rail %d is ACTIVE\n",i);
```

3. The following program extract requests the names of the members of the MEMORY CHANNEL API cluster, the number of logical rails, and the active MEMORY CHANNEL logical rails bitmask.

```
imc_railinfo mask;
imc_hostinfo hostinfo;
unsigned     nrails;

int status;
imc_infoType items[4];

items[0] = IMC_GET_ACTIVERAILS;
items[1] = IMC_GET_HOSTS;
items[2] = IMC_GET_NRAILS;
items[3] = 0;

status = imc_getclusterinfo(items,3,mask,sizeof(imc_railinfo),\
            &hostinfo,sizeof(imc_hostinfo),&nrails,sizeof(unsigned));
```

RETURN VALUES

The `imc_getclusterinfo( )` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_BADPARM`	An invalid parameter was specified in the call to the `imc_getclusterinfo( )` function.
`IMC_NOTINIT`	This host has not been initialized to use the MEMORY CHANNEL API library.

SEE ALSO

Introduction: `imc`(3)

Commands: `imc_init`(1), `imcs`(1)

Functions: `imc_api_init`(3), `imc_kill`(3), `imc_wait_cluster_event`(3)

NAME

`imc_kill` – Sends a signal to a running process

SYNOPSIS

```
#include <sys/imc.h>
int imc_kill(
        char * hostname,
        pid_t pid,
        int signal);
```

PARAMETERS

hostname	Specifies the host on which the target process is executing.
pid	Specifies the process identifier (PID) of the target process.
signal	Specifies the signal to be sent to the target process.

DESCRIPTION

The `imc_kill( )` function sends a signal to a target process that is executing on the MEMORY CHANNEL API cluster member specified by the *hostname* parameter. A list of valid host names can be obtained by calling the `imc_getclusterinfo( )` function. The PID for the target process is specified by the *pid* parameter and it must be a valid PID. Zero and negative PID values are not valid.

Processes that are executing with root privileges are not allowed to send signals across the MEMORY CHANNEL API cluster.

The `imc_kill( )` function is similar to the UNIX `kill`(2) function; however, it does not support the sending of signals to multiple processes.

RETURN VALUES

The `imc_kill` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.

IMC_BADHOST	The host name specified in the *hostname* parameter is invalid.
IMC_BADPARM	The value specified in the *pid* parameter is invalid.
IMC_EINVAL	The signal specified in the *signal* parameter is not a valid signal number. Zero and negative PID values are not permitted.
IMC_EINVAL	The *signal* parameter is SIGKILL, SIGSTOP, SIGTSTP, or SIGCONT and the PID specified in the *pid* parameter is 1 (proc1).
IMC_EPERM	The real or saved user ID does not match the real or effective user ID of the receiving process, the calling process does not have appropriate privilege, and the process is not sending a SIGCONT signal to one of its session's processes.
IMC_ESRCH	No process can be found corresponding to that specified by the *pid* parameter.
IMC_NOROOT	Superuser signalling across the MEMORY CHANNEL API cluster is not permitted.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Function: imc_api_init(3), imc_getclusterinfo(3), kill(2)

NAME

imc_lkacquire – Acquires a lock from an existing set of locks

SYNOPSIS

```
#include <sys/imc.h>
int imc_lkacquire(
          imc_lkid_t lock,
          int index,
          int flag,
          int trylock);
```

PARAMETERS

lock
Identifies the lock set from which the lock is to be acquired. The lock set specified by the *lock* parameter must previously have been allocated by a call to the imc_lkalloc( ) function.

index
Specifies the lock to be acquired. The value of the *index* parameter is in the range zero (0) to *(count-1)*, where *count* is the value returned by the imc_lkalloc( ) function when it created the lock set.

flag
This parameter is reserved for future use by DIGITAL. You must set the value of this parameter to zero (0).

trylock
Specifies whether the imc_lkacquire function should return immediately if the lock is busy or wait until it can acquire the lock. The *trylock* parameter contains one of the following values:

IMC_LOCKWAIT
Wait until the lock becomes free and then acquire the lock before returning.

IMC_LOCKNOWAIT
Return immediately if the lock is in use.

DESCRIPTION

The `imc_lkacquire( )` function tries to acquire the lock specified by the *index* parameter from the lock set specified in the *lock* parameter. If the lock is in use, the function can wait until the lock becomes free, or it can return immediately without acquiring the lock. The return values for the function indicate whether or not the lock was successfully acquired.

When a process acquires a lock, no other process executing on the MEMORY CHANNEL API cluster can acquire that lock.

Waiting for busy locks to become free entails busy spinning and has a significant effect on performance. Therefore, in the interest of overall system performance, applications should acquire locks only as they are needed and release them promptly.

If a system failure occurs on a host on which a process that holds a lock is executing, all locks associated with the host are automatically released.

All locks acquired by a process are automatically released when the process exits.

It is illegal for a process to acquire locks recursively. If a process acquires a lock that it has already acquired and not released, an error will occur. The correct sequence is for the process to acquire the lock, release it, and then acquire it again.

RETURN VALUES

The `imc_lkacquire` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_BADLOCK`	Either the lock set specified by the *lock* parameter or the lock specified by the *index* parameter is out of range.
`IMC_BADPARM`	An invalid parameter was specified in the call to the `imc_lkacquire( )`function.
`IMC_CORRUPTLOCK`	An attempt was made to acquire a lock from an invalid or corrupted lock set.

IMC_LOCKPRIOR	The process attempted to acquire a lock that it already holds.
IMC_NOLOCKGOT	The imc_lkacquire( ) function returned without gaining ownership of the lock.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_lkalloc(3), imc_lkdealloc(3), imc_lkrelease(3)

NAME

imc_lkalloc – Creates a lock set

SYNOPSIS

```
#include<sys/imc.h>
int imc_lkalloc(
          imc_key_t key,
          int *count,
          imc_perm_t perm,
          int flag,
          imc_lkid_t *lock);
```

PARAMETERS

key — Identifies the lock set to be allocated.

count — Specifies the number of locks created in the lock set, and returns the number of locks actually created.

perm — Specifies the access permission for the lock set. The permission code is similar to the UNIX permission convention, except that there is no execute flag. The value of the *perm* parameter is obtained by carrying out a logical OR operation on the following values:

IMC_LKU	User access to locks
IMC_LKG	Group access to locks
IMC_LKO	Other access to locks

flag — Specifies the creation flag for the lock set. The *flag* parameter has one of the following values:

IMC_CREATOR	If the lock set does not already exist on the MEMORY CHANNEL API cluster, allocate the lock set and atomically acquire the

		first lock (that is, lock zero (0)) in the set. If the IMC_CREATOR flag is specified for a lock set that already exists, an error condition will result.
	ZERO (0)	Allocate the lock set without attempting to acquire the first lock in the set.
lock		Returns a value that uniquely identifies the allocated lock set. If the imc_lkalloc( ) function fails to allocate a lock set, the value of the *lock* parameter is set to NULL.

DESCRIPTION

The imc_lkalloc( ) function creates a set of locks that enable applications to coordinate access to clusterwide resources. The number of locks in the lock set is specified by the *count* parameter. The maximum number of locks that a set can contain is specified by the IMC_MAXNUMLOCKS value in the MEMORY CHANNEL API library header file.

The imc_lkalloc( ) function provides a feature that allows a process to atomically (that is, in a single operation) allocate a lock set and acquire the first lock in the set. This feature can be used to coordinate application initialization in a MEMORY CHANNEL API cluster. To atomically allocate the lock set and acquire the first lock, specify the value IMC_CREATOR for the *flag* parameter.

The method for establishing a relationship between a lock and a resource is application-specific, and is beyond the scope of the MEMORY CHANNEL API library.

All lock sets allocated by a process are automatically deallocated when the process exits.

RETURN VALUES

The imc_lkalloc function returns one of the following values:

IMC_SUCCESS	Normal successful completion.
IMC_BADPARM	An invalid parameter was specified in the call to the `imc_lkalloc( )`function.
IMC_BADSIZE	The lock set is already allocated on the MEMORY CHANNEL API cluster, and the size of the set as specified in this call to the `imc_lkalloc( )`function does not match the size specified in the previous call.
IMC_EXISTS	The lock set already exists on the MEMORY CHANNEL API cluster.
IMC_MAPENTRIES	An attempt has been made to exceed the maximum number of process map entries. This maximum is set by the *vm_mapentries* parameter.
IMC_MCFULL	There is not enough MEMORY CHANNEL address space available to allocate the lock set.
IMC_NOMEM	There is insufficient local memory available to allocate the lock set.
IMC_NORESOURCES	There are insufficient MEMORY CHANNEL data structures available to allocate the lock set.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.
IMC_PERMIT	The lock set is already allocated, with a permission code that is incompatible with the code specified in the *perm* parameter.
IMC_PRIOR	The lock set has already been allocated by the calling process.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_lkacquire(3), imc_lkdealloc(3), imc_lkrelease(3)

NAME

imc_lkdealloc – Deallocates a lock set

SYNOPSIS

```
#include <sys/imc.h>
int imc_lkdealloc(
        imc_lkid_t *lock);
```

PARAMETER

lock	Identifies the lock set to be deallocated. The lock set specified by the *lock* parameter must previously have been allocated by a call to the imc_lkalloc( ) function.

DESCRIPTION

The imc_lkdealloc( ) function deallocates the lock set specified by the *lock* parameter. An attempt to deallocate a lock set that contains active locks will result in an error condition.

All lock sets allocated by a process are automatically deallocated when the process exits.

RETURN VALUES

The imc_lkdealloc function returns one of the following values:

IMC_SUCCESS	Normal successful completion.
IMC_BADLOCK	The lock set specified by the *lock* parameter does not exist.
IMC_BADPARM	An invalid parameter was specified in the call to the imc_lkdealloc( )function.
IMC_CORRUPTLOCK	An attempt was made to deallocate an invalid or corrupted lock set.

IMC_LOCKACTIVE	The lock set has active locks.
IMC_NOTALLOC	The lock set is not allocated.
IMC_NOTINIT	This host has not been initialized to use the MEMORY CHANNEL API library.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_lkacquire(3), imc_lkalloc(3), imc_lkrelease(3)

NAME

`imc_lkrelease` – Releases a lock in a lock set

SYNOPSIS

```
#include <sys/imc.h>
int imc_lkrelease(
          imc_lkid_t lock,
          int index);
```

PARAMETERS

lock	Identifies the lock set that contains the lock to be released. The lock set specified by the *lock* parameter must previously have been allocated by a call to the `imc_lkalloc( )` function.
index	Specifies the lock to be released. The value of the *index* parameter is in the range zero (0) to *(count-1)*, where *count* is the value returned by the `imc_lkalloc( )` function when it created the lock set.

DESCRIPTION

The `imc_lkrelease( )` function releases a lock that is being held as a result of a call to the `imc_lkacquire( )` function or the `imc_lkalloc( )` function.

If the lock specified by the *lock* and *index* parameters is not being held, an error condition will result.

If a system failure occurs on a host on which a process that holds a lock is executing, all locks associated with the host are automatically released.

All locks acquired by a process are automatically released when the process exits.

RETURN VALUES

The `imc_lkrelease` function returns one of the following values:

`IMC_SUCCESS`	Normal successful completion.
`IMC_BADLOCK`	Either the lock set specified by the *`lock`* parameter or the lock specified by the *`index`* parameter does not exist.
`IMC_CORRUPTLOCK`	An attempt was made to release a lock in an invalid or corrupted lock set.
`IMC_LOCKNOTHELD`	The lock that the `imc_lkrelease( )` function tried to release was not being held.

SEE ALSO

Introduction: `imc`(3)

Commands: `imc_init`(1), `imcs`(1)

Functions: `imc_api_init`(3), `imc_lkacquire`(3), `imc_lkalloc`(3), `imc_lkdealloc`(3)

NAME

`imc_perror` – Prints a message that explains a MEMORY CHANNEL function error

SYNOPSIS

```
#include <sys/imc.h>
void imc_perror(
          char *userstring,
          int code);
```

PARAMETERS

userstring	Specifies a string to be prefixed to the error message.
code	Specifies the return status from the MEMORY CHANNEL API library function that failed.

DESCRIPTION

The `imc_perror( )` function prints a message to standard error output that gives an explanation of the error status specified in the *code* parameter. The message is made up of the following:

- The prefix specified in the *userstring* parameter
- A colon (:)
- A blank space
- The error message
- A newline character

RETURN VALUES

The `imc_perror` function does not return any values.

SEE ALSO

Introduction: `imc`(3)

imc_perror(3)

Commands: `imc_init`(1), `imcs`(1)

Functions: `imc_api_init`(3), `imc_ckerrcnt_mr`(3)

NAME

imc_rderrcnt – Reads the total error count across all logical rails

SYNOPSIS

```
#include <sys/imc.h>
int imc_rderrcnt( void );
```

PARAMETERS

The imc_rderrcnt( ) function does not take any parameters.

DESCRIPTION

Note

DIGITAL recommends that you use the imc_rderrcnt_mr( ) function rather than the imc_rderrcnt( ) function.

The imc_rderrcnt( ) function reads the total error count across all logical rails and returns the value to the calling program. The total error count is updated whenever a MEMORY CHANNEL error occurs. The count is not guaranteed to be up to date with the most recent MEMORY CHANNEL transfer. However, you can use it with the imc_ckerrcnt( ) function to determine whether any errors occurred since the last time the count was updated.

The MEMORY CHANNEL hardware guarantees that no corrupt data will be written to host systems, and that all data will be delivered to the host systems in the sequence in which the data is written to the MEMORY CHANNEL hardware. The atomic unit of transfer on MEMORY CHANNEL is 32 bits. Statistically, the error rate of the MEMORY CHANNEL hardware is of the order of three errors per year.

If an error count is being updated at the time the imc_rderrcnt( ) function is called, the function returns a negative value. Programs should check for this eventuality and call the function again to ensure that it reads the correct error count.

You can use the imc_rderrcnt( ) function along with the imc_ckerrcnt( ) function to construct application-specific error-detection protocols.

RETURN VALUES

On successful completion, the imc_rderrcnt( ) function returns a positive integer that contains the total error count.

The imc_rderrcnt( ) function returns a negative value if the error count is being updated when the function is called.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_ckerrcnt(3), imc_ckerrcnt_mr(3), imc_perror(3), imc_rderrcnt_mr(3)

NAME

imc_rderrcnt_mr – Reads the clusterwide error count for the specified logical rail

SYNOPSIS

```
#include <sys/imc.h>
int imc_rderrcnt(
        int logical_rail);
```

PARAMETER

logical_rail	Specifies the MEMORY CHANNEL logical rail for which the error count is to be read. The first logical rail is numbered zero (0), the second logical rail is numbered 1, and so on, up to a maximum defined by a constant, IMC_MAXRAILS.

DESCRIPTION

The imc_rderrcnt_mr( ) function reads the total error count for the specified logical rail and returns the value to the calling program. The error count is updated whenever a MEMORY CHANNEL error occurs. The count is not guaranteed to be up to date with the most recent MEMORY CHANNEL transfer. However, you can use it with the imc_ckerrcnt_mr( ) function to determine whether any errors occurred since the last time the count was updated.

The MEMORY CHANNEL hardware guarantees that no corrupt data will be written to host systems, and that all data will be delivered to the host systems in the sequence in which the data is written to the MEMORY CHANNEL hardware. The atomic unit of transfer on MEMORY CHANNEL is 32 bits. Statistically, the error rate of the MEMORY CHANNEL hardware is of the order of three errors per year.

If the error count is being updated at the time the imc_rderrcnt_mr( ) function is called, the function returns a negative value. Programs should check for this eventuality and call the function again to ensure that it reads the correct error count.

You can use the imc_rderrcnt_mr( ) function along with the imc_ckerrcnt_mr( ) function to construct application-specific error-detection protocols.

RETURN VALUES

On successful completion, the imc_rderrcnt_mr function returns a positive integer that contains the error count for the specified logical rail.

The imc_rderrcnt_mr( ) function returns a negative value if the error count is being updated when the function is called.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_ckerrcnt_mr(3), imc_perror(3)

NAME

imc_wait_cluster_event – Blocks the caller until a MEMORY CHANNEL API cluster event occurs

SYNOPSIS

```
#include <sys/imc.h>
int imc_wait_cluster_event(
        imc_eventType *i_eventType,
        unsigned long i_numEvents,
        unsigned long i_unused,
        [void *io_current_config,
        unsigned long i_current_config_length]...);
```

PARAMETERS

i_eventType Points to a data structure that contains information about valid MEMORY CHANNEL API cluster events for which to wait. The *i_eventType* parameter has the following values:

IMC_CC_EVENT_HOST	A host has joined the MEMORY CHANNEL API cluster, or a host has been removed from the MEMORY CHANNEL API cluster.
IMC_CC_EVENT_RAIL	The logical rail configuration of the MEMORY CHANNEL API cluster has changed; a logical rail has come on line, or a logical rail has gone off line.

Note

A MEMORY CHANNEL API cluster is formed when a number of hosts are physically connected by a MEMORY CHANNEL interconnect, and when each

host has invoked the imc_init(1) command.

i_numEvents Specifies the number of events in the data structure identified by the *i_eventType* parameter. This parameter must have a value of 1 or greater.

i_unused This parameter is currently unused. You must set the value of this parameter to zero (0).

io_current_config Points to a data structure that contains information about the MEMORY CHANNEL API cluster configuration item being monitored. There are two valid data structure types, as follows:

imc_hostinfo	If the *i_eventType* parameter has the value IMC_CC_EVENT_HOST, the *io_current_config* parameter points to a structure of type imc_hostinfo, which is returned by the imc_getclusterinfo( ) function.
imc_railinfo	If the *i_eventType* parameter has the value IMC_CC_EVENT_RAIL, the *io_current_config* parameter points to a structure of type imc_railinfo, which is returned by the imc_getclusterinfo( ) function.

If a value of NULL is specified for the *io_current_config* parameter, the

`imc_wait_cluster_event( )` function will use the current internal value of the MEMORY CHANNEL API cluster configuration item being monitored. On return, the user must access this value using the `imc_getclusterinfo( )` function.

i_current_config_length

Specifies the length of the *io_current_config* parameter. If the *io_current_config* parameter has a value of `NULL`, the *i_current_config_length* parameter is ignored.

DESCRIPTION

The `imc_wait_cluster_event( )` function blocks the calling entity until a specified MEMORY CHANNEL API cluster event occurs.

Note

The `imc_wait_cluster_event( )` function waits for MEMORY CHANNEL API cluster events, not Production Server cluster events.

Two MEMORY CHANNEL API cluster events are valid:

- A host joins or leaves the MEMORY CHANNEL API cluster.
- The logical rail configuration of the MEMORY CHANNEL API cluster changes.

At least one MEMORY CHANNEL API cluster event must be specified in the call to the `imc_wait_cluster_event( )` function; more than one MEMORY CHANNEL API cluster event may be specified.

The `imc_wait_cluster_event( )` function initially checks the current representation of the MEMORY CHANNEL API cluster configuration event being monitored.

On return, the *io_current_config* parameter points to the new MEMORY CHANNEL API cluster configuration.

If the *io_current_config* parameter has been set to a value of NULL, the imc_wait_cluster_event( ) function will use the current internal value of the MEMORY CHANNEL API cluster configuration item being monitored; and the *i_current_config_length* parameter will be ignored. If this is the case, the user must access the value of the MEMORY CHANNEL API cluster configuration item on return, using the imc_getclusterinfo( ) function.

EXAMPLES

1. The following program extract requests the names of the members of the MEMORY CHANNEL API cluster using the imc_getclusterinfo( ) function, and then calls the imc_wait_cluster_event function to wait for a configuration change to be notified:

```
imc_hostinfo    hostinfo;
int             status;
imc_infotype    items[2];
imc_eventType   events[2];

items[0] = IMC_GET_HOSTS;
items[1] = 0;

events[0] = IMC_CC_EVENT_HOSTS;
events[1] = 0;

status =
       imc_getclusterinfo(items,1,&hostinfo,sizeof(imc_hostinfo));

if (status != IMC_SUCCESS)
    imc_perror("imc_getclusterinfo:",status);

status =
       imc_wait_cluster_event(events,1,0,&hostinfo,sizeof(imc_hostinfo));

if (status != IMC_HOST_CHANGE)
    imc_perror("imc_wait_cluster_event didn't complete:",status);
```

2. The following program extract requests the names of the members of the MEMORY CHANNEL API cluster and the active MEMORY CHANNEL logical rails bitmask, and then waits for an event change on either:

```
imc_railinfo    mask;
imc_hostinfo    hostinfo;

int             status;
imc_infoType    items[3];
imc_eventType   events[3];
```

```
items[0] = IMC_GET_ACTIVERAILS;
items[1] = IMC_GET_HOSTS;
items[2] = 0;

events[0] = IMC_CC_EVENT_RAILS;
events[1] = IMC_CC_EVENT_HOSTS;
events[2] = 0;

status = imc_getclusterinfo(items,2,mask,sizeof(imc_railinfo),
                                &hostinfo,sizeof(imc_hostinfo));

if (status != IMC_SUCCESS)
    imc_perror("imc_getclusterinfo:",status);

status = imc_wait_cluster_event(events, 2, 0, \\
                                mask, sizeof(imc_railinfo),
                                &hostinfo, sizeof(imc_hostinfo));

if ((status != IMC_HOST_CHANGE) && (status != IMC_RAIL_CHANGE))
    imc_perror("imc_wait_cluster_event didn't complete:",status);
```

3. The following program extract waits for an event change on either the members of the MEMORY CHANNEL API cluster or the active MEMORY CHANNEL logical rails:

```
int status;
imc_eventType events[3];

events[0] = IMC_CC_EVENT_HOSTS;
events[1] = IMC_CC_EVENT_RAILS;
events[2] = 0;

status = imc_wait_cluster_event(events, 2, 0, NULL, 0, NULL, 0);

if ((status != IMC_HOST_CHANGE) && (status != IMC_RAIL_CHANGE))
    imc_perror("imc_wait_cluster_event didn't complete:",status);
```

RETURN VALUES

The `imc_wait_cluster_event` function returns one of the following values:

`IMC_NOTINIT`	This host has not been initialized to use the MEMORY CHANNEL API library.

IMC_BADPARM	An invalid parameter was specified in the call to the imc_wait_cluster_event( ) function.
IMC_INTR	The imc_wait_cluster_event( ) function was interrupted by a signal.
IMC_HOST_CHANGE	The host configuration has changed.
IMC_RAIL_CHANGE	The logical rail configuration has changed.
IMC_MULTIPLE_CHANGE	More than one of the monitored configuration items has changed.

SEE ALSO

Introduction: imc(3)

Commands: imc_init(1), imcs(1)

Functions: imc_api_init(3), imc_getclusterinfo(3)

A

Frequently Asked Questions

This appendix contains answers to questions asked by programmers who use the MEMORY CHANNEL API to develop programs for TruCluster systems.

A.1 IMC_NOMAPPER Return Code

Question: An attempt was made to do an attach to a coherent region using the `imc_asattach` function. The function returned the value `IMC_NOMAPPER`. What does this mean?

Answer: This return value indicates that the `imc_mapper` process is missing from a system in your MEMORY CHANNEL API cluster.

The `imc_mapper` process is started automatically in the following cases:

- On system initialization, when the configuration variable `IMC_AUTO_INIT` has a value of 1. (See Chapter 1 for more information about the `IMC_AUTO_INIT` variable.)
- When you execute the `imc_init` command for the first time.

To solve this problem, reboot the system from which the `imc_mapper` process is missing.

This error may occur if you shut down a system to single-user mode from `init` level 3, and then return the system to multi-user mode without doing a complete reboot. If you want to reboot a system that runs TruCluster MEMORY CHANNEL software, you must do a full reboot of that system.

A.2 Efficient Data Copy

Question: How can data be copied to a MEMORY CHANNEL transmit region in order to obtain maximum MEMORY CHANNEL bandwidth?

Answer: The MEMORY CHANNEL API `imc_bcopy` function provides an efficient way of copying aligned or unaligned data to MEMORY CHANNEL. The `imc_bcopy` function has been optimized to make maximum use of the buffering capability of a DIGITAL Alpha CPU.

You can also use the `imc_bcopy` function to copy data efficiently between two buffers in standard memory.

A.3 MEMORY CHANNEL Bandwidth Availability

Question: Is maximum MEMORY CHANNEL bandwidth available when using coherent MEMORY CHANNEL regions?

Answer: No. Coherent regions use the loopback feature to ensure local coherency. Therefore, every write data cycle has a corresponding cycle to loop the data back; this halves the available bandwidth. See Section 2.2.1.3 for more information about the loopback feature.

A.4 MEMORY CHANNEL API Cluster Configuration Change

Question: How can a program determine whether a MEMORY CHANNEL API cluster configuration change has occurred?

Answer: The new `imc_wait_cluster_event( )` function can be used to monitor hosts joining or leaving the MEMORY CHANNEL API cluster, or to monitor changes in the state of the active logical rails. You can write a program that calls the `imc_wait_cluster_event( )` function in a separate thread; this blocks the caller until a state change occurs.

A.5 Bus Error Message

Question: When a program tries to set a value in an attached transmit region, it crashes with the following message:

```
Bus error (core dumped)
```

Why does this happen?

Answer: The data type of the value may be smaller than 32 bits (in C, an `int` is a 32–bit data item, and a `short` is a 16–bit data item). The DIGITAL Alpha processor, like other RISC processors, reads and writes data in 64–bit units or 32–bit units. When you assign a value to a data item that is smaller than 32 bits, the compiler generates code that loads a 32–bit unit, changes the bytes that are to be modified, and then stores the entire 32–bit unit. If such a data item is in a MEMORY CHANNEL region attached for transmit, the assignment causes a read operation to occur in the attached area. Because transmit areas are write-only, a bus error is reported.

You can prevent this problem by ensuring that all accesses are done on 32–bit data items. See Section 2.2.3 for more information.

A.6 Deciding Which TruCluster Product To Use

Question: There are three TruCluster products. Which product should I use?

Answer: If your application requires high availability and access to disks on shared SCSI buses, and uses the Distributed Lock Manager (DLM) or Distributed Raw Disk (DRD) services, you should use the TruCluster Production Server product. You need MEMORY CHANNEL interconnect hardware and shared SCSI buses. Applications which require these features include certain database management systems. (The TruCluster Production Server product combines the features of the TruCluster Available Server product and the TruCluster MEMORY CHANNEL Software product.)

If your application requires high availability with management of failover of critical services to a backup system, you need the TruCluster Available Server product. You need shared SCSI buses, but the MEMORY CHANNEL interconnect is not supported. NFS is an example of a highly available service managed by TruCluster Available Server.

If your application requires a high bandwidth, low latency interconnect, you need the TruCluster MEMORY CHANNEL Software product. You need MEMORY CHANNEL interconnect hardware, but shared SCSI buses are not supported. The DIGITAL Parallel Software Environment (PSE) product is an example of an application that requires these features.

A.7 Finding Out More About MEMORY CHANNEL

Question: Where can I get more information about MEMORY CHANNEL?

Answer: You can find out more about MEMORY CHANNEL from the DIGITAL High Performance Technical Computing Web server, at the following URL:

`http://www.digital.com/info/hpc`

You may also mail your questions to: `high-performance@digital.com`

Index

A

accessing MEMORY CHANNEL addresses, 2–2
address mapping
  defined, 2–2
  how to implement, 2–2
application development models, 2–24
attach
  broadcast mode, 2–3, 2–4
  loopback mode, 2–3, 2–6, A–2
  point-to-point mode, 2–3, 2–5

B

broadcast attach, 2–4
building applications
  compiling, 3–2
  header files, 3–1
  library, 3–1

C

clu_install script, 1–7
clu_ivp utility
  using to detect configuration errors, 1–11
cluster information functions, 2–22
cluster signals, 2–22
clusterwide address space, 2–2
clusterwide locks
  allocating, 2–21
  defined, 2–19
  example, 2–19e
  performance impact, 2–22
  single-threaded access, 2–21
coherency
  initial, 2–7, 2–8
  latency related, 2–11
compiling applications, 3–2
console
  boot_reset variable, 1–3
  bus_probe_algorithm variable, 1–3
  error messages, 1–19

D

deinstalling MEMORY CHANNEL subsets, 1–4
doconfig program
  kernel configuration file, 1–8

E

error management, 2–14
  using imc_ckerrcnt_mr, 2–17
    example, 2–18e
  using imc_rderrcnt_mr, 2–17
    example, 2–16e
/etc/hosts
  entry for IP name, 1–7
  updated by installation, 1–3
/etc/rc.config
  IMC_AUTO_INIT variable, 1–12
  updated by installation, 1–3
/etc/sysconfigtab
  rm_no_inheritance attribute, 1–19
  setting rm_rail_style parameter, 1–15
  using sysconfigdb(8) to amend, 1–2, 1–16
  vm-mapentries parameter, 1–18

vm-syswiredpercent parameter, 1–17, 1–20

F

fatal error
rm_check_cables, 1–21
rm_delete_context, 1–20
rm_slave_init, 1–21

G

/genvmunix
booting after installing new hardware, 1–5n
booting to single-user mode before deinstalling subsets, 1–4
booting when upgrading DIGITAL UNIX Version 4.0, 1–5

H

handling errors, 2–14
using imc_ckerrcnt_mr, 2–17
example, 2–18e
using imc_rderrcnt_mr, 2–17
example, 2–16e
hardware configuration, 1–2
header files, 3–1

I

imc introduction, 3–3
imc_api_init function, 2–1, 2–10, 3–16
imc_asalloc function, 2–10, 2–13, 3–18
key parameter, 2–2
imc_asattach function, 2–10, 2–11, 2–14, 3–23, A–1
dir parameter, 2–2
imc_asattach_ptp function, 2–2, 3–29
imc_asdealloc function, 2–11, 3–33
imc_asdetach function, 2–11, 3–35
IMC_AUTO_INIT variable, 1–12, A–1
imc_bcopy function, 2–8, 3–37, A–1
imc_ckerrcnt function, 3–42
imc_ckerrcnt_mr function, 2–15, 3–44
IMC_EXISTS
return status, 2–21
imc_getclusterinfo
example, 2–23e
imc_getclusterinfo function, 2–22, 3–46
imc_init command, 1–12, 1–22, 2–2, 3–8, A–1
IMC_NOTINIT return status, 1–18
maxalloc parameter, 1–17
maxrecv parameter, 1–17
imc_kill function, 2–22, 3–50
imc_lkacquire function, 2–19, 2–21, 3–52
imc_lkalloc function, 2–19, 3–55
imc_lkdealloc function, 3–59
imc_lkrelease function, 2–21, 3–61
IMC_MAPENTRIES return status, 1–18, 1–22
imc_mapper process
not present, A–1
IMC_MAX_ALLOC variable, 1–17
IMC_MAX_RECV variable, 1–17, 1–20
IMC_MCFULL return status, 1–22
IMC_NOMAPPER return status, A–1
IMC_NOMEM return status, 1–23
IMC_NORESOURCES return status, 1–23
IMC_NOTINIT return status, 1–18
imc_perror function, 2–10, 3–63
imc_rderrcnt function, 3–65

imc_rderrcnt_mr function, 2–15, 3–67
IMC_RXFULL return status, 1–17, 1–22
imc_wait_cluster_event
  example, 2–23e
imc_wait_cluster_event function, 2–22, 3–69, A–2
IMC_WIRED_LIMIT return status, 1–17, 1–22
imcs command, 2–24, 3–11
initial coherency, 2–7, 2–8
initialization failure
  cables not attached, 1–19
  check hardware configuration, 1–20
  node running software Version 1.4, 1–20
  rm_no_inheritance not set to 1, 1–19
  vm-syswiredpercent parameter, 1–20
initializing MEMORY CHANNEL API library, 1–12, 2–1
installation
  restrictions, 1–2
installation verification procedure (*See clu_ivp utility*)
IP name and address
  assigning during installation, 1–2
  Host number 42 reserved, 1–3n
  overriding existing, 1–7
  specifying IP name for MEMORY CHANNEL adapter, 1–7

K

kernel
  building after changing adapter, 1–5n
  building after installing new hardware, 1–5n
  building new, 1–8
  editing kernel configuration file, 1–8
  moving to root directory, 1–9
  specifying kernel configuration file, 1–8

L

latency related coherency, 2–11
library, 3–1
license
  registering, 1–5
lmf command
  list, 1–4
  register, 1–6
  reset, 1–4
locks
  allocating, 2–21
  defined, 2–19
  example, 2–19e
  performance impact, 2–22
  single-threaded access, 2–21
logical rail failure, 1–16n
loopback attach, 2–6, A–2

M

malloc
  failure, 1–23
MEMORY CHANNEL
  configuration tuning, 1–16
  error rates, 2–14
  multirail model, 1–13
  troubleshooting, 1–18
MEMORY CHANNEL address space
  extending, 1–22
MEMORY CHANNEL API cluster, 1–1n
MEMORY CHANNEL API library
  compiling, 3–2
  developing applications, 2–1
  header files, 3–1
  initializing, 1–12, 2–1

library, 3–1
MEMORY CHANNEL API library commands
imc_init, 1–12, 1–18, 2–2, A–1
imcs, 2–24
MEMORY CHANNEL API library functions
imc_api_init, 2–1, 2–10
imc_asalloc, 2–2, 2–10, 2–13
imc_asattach, 2–2, 2–10, 2–11, 2–14, A–1
imc_asattach_ptp, 2–2
imc_asdealloc, 2–11
imc_asdetach, 2–11
imc_bcopy, 2–8, A–1
imc_ckerrcnt_mr, 2–15
imc_getclusterinfo, 2–22
imc_kill, 2–22
imc_lkacquire, 2–19, 2–21
imc_lkalloc, 2–19
imc_lkrelease, 2–21
imc_perror, 2–10
imc_rderrcnt_mr, 2–15
imc_wait_cluster_event, 2–22, A–2
MEMORY CHANNEL region
allocating, 2–10, 2–13
attaching for transmit, 2–10, 2–14
detaching, 2–11
message passing and shared memory
comparison, 2–24
multirail model, 1–13
default style, 1–14
failover pair, 1–14
logical rail, 1–13
physical rail, 1–13
rm_rail_style parameter, 1–15
single-rail, 1–13

O

operating system
subsets, 1–4

P

PAK, 1–5
point-to-point attach, 2–5
process virtual memory
extending, 1–23
Product Authorization Key (*See PAK*)
Production Server cluster, 1–1n

R

reading from transmit area
segmentation violation, A–2
reading from transmit pointer
segmentation violation, 2–8
rebooting
after installation, 1–9
receive address space
extending, 1–22
resource limitations
IMC_MAPENTRIES return status, 1–18, 1–22
IMC_MCFULL return status, 1–22
IMC_NOMEM return status, 1–23
IMC_NORESOURCES return status, 1–23
IMC_RXFULL return status, 1–17, 1–22
IMC_WIRED_LIMIT return status, 1–17, 1–22
rm_check_cables
fatal error, 1–21
rm_delete_context
fatal error, 1–20
rm_no_inheritance attribute
not set to 1, 1–19
rm_rail_style parameter
possible values, 1–15
rm_slave_init
fatal error, 1–21

S

segmentation violation
  caused by reading from transmit area, A–2
  caused by reading from transmit pointer, 2–8
setld command
  deleting subsets, 1–4
  loading MEMORY CHANNEL kit, 1–6
shared memory and message passing
  comparison, 2–24
signals, 2–22
subsets
  installing, 1–6
  operating system, 1–4

T

troubleshooting
  initialization failure, 1–19
tuning MEMORY CHANNEL, 1–16

U

/usr/var/adm
  messages file, 1–19

V

virtual memory map entries
  extending, 1–22
vm-mapentries parameter, 1–22
vm-syswiredpercent parameter, 1–20, 1–22

W

wired memory limit
  extending, 1–22

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 *or* U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader's Comments

TruCluster Production Server Software
MEMORY CHANNEL Application Programming Interfaces
AA-QTN4C-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 884-0120, Attn: UBPG Publications, ZKO3-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Usability (ability to access information quickly)	☐	☐	☐	☐

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____
_____
_____
_____

What version of the software described by this manual are you using? _____

Name, title, department _____
Mailing address _____
Electronic mail _____
Telephone _____
Date _____

--------- Do Not Cut or Tear - Fold Here and Tape ----------

digital™

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UBPG PUBLICATIONS MANAGER
ZKO3 3/Y32
110 SPIT BROOK RD
NASHUA NH 03062 9987

--------- Do Not Cut or Tear - Fold Here ----------

Cut on Dotted Line